IBM Gekko RISC Microprocessor User's Manual

Version 1.2

May 25, 2000

IBM Confidential

Trademarks

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM	IBM Logo	PowerPC
AIX	PowerPC 750	Gekko

Other company, product, and service names may be trademarks or service marks of others.

Document History	**Date**	**Description**
Preliminary Edition	3/29/00	Initial release of new format
2nd Preliminary Edition	4/18/00	Minor changes, most are transparent to user (removal of conditional text, etc.)
3rd Preliminary Edition	5/25/00	Minor change in section 4.5.6

This **unpublished document** is the preliminary edition of IBM Gekko RISC Microprocessor User's Manual.

This document contains information on a new product under development by IBM. IBM reserves the right to change or discontinue this product without notice.

CONTENTS

Chapter 1 Gekko Overview

Chapter 2 Programming Model

CONTENTS (Continued)

CONTENTS (Continued)

CONTENTS (Continued)

Chapter 4 Exceptions

Chapter 5 Memory Management

CONTENTS (Continued)

Chapter 7 Signal Descriptions

Chapter 9 L2 Cache, Locked D-Cache, DMA and Write Gather Pipe

Chapter 10 Power and Thermal Management

ILLUSTRATIONS

Chapter 1—Gekko Overview

Chapter 2—Programming Model

Chapter 3—Gekko Instruction and Data Cache Operation

Chapter 4—Exceptions

ILLUSTRATIONS (Continued)

Chapter 5—Memory Management

Chapter 6—Instruction Timing

Chapter 7—Signal Descriptions

Chapter 8—Bus Interface Operation

ILLUSTRATIONS (Continued)

Chapter 10—Power and Thermal Management

Chapter 11— Performance Monitor

Chapter 12—Instruction Set

TABLES

Chapter 1—Gekko Overview

Chapter 2—Programming Model

TABLES (Continued)

Chapter 3—Gekko Instruction and Data Cache Operation

Chapter 4—Exceptions

Chapter 5—Memory Management

Chapter 6—Instruction Timing

Chapter 7—Signal Descriptions

Chapter 8—Bus Interface Operation

Chapter 9—L2 Cache, Locked D-Cache, DMA and Write Gather Pipe

Chapter 10—Power and Thermal Management

Chapter 11— Performance Monitor

Chapter 12—Instruction Set

Appendix A—– Gekko Instruction Set

Chapter 1 Gekko Overview

Gekko is an implementation of the PowerPC architecture with enhancements to improve the floating point performance and the data transfer capability. This chapter provides an overview of the PowerPC Gekko microprocessor features, including a block diagram showing the major functional components. It also provides information about how Gekko implementation complies with the PowerPC™ architecture definition .

1.1 Gekko Microprocessor Overview

This section describes the features and general operation of Gekko and provides a block diagram showing major functional units. Gekko is an implementation of the PowerPC microprocessor family of reduced instruction set computer (RISC) microprocessors with extensions to improve the floating poing performance. Gekko implements the 32-bit portion of the PowerPC architecture, which provides 32-bit effective addresses, integer data types of 8, 16, and 32 bits, and floating-point data types of single- and double-precision. Gekko extends the PowerPC architecture with the paired single-precision floating point data type and a set of paired single floating point instructions. Gekko is a superscalar processor that can complete two instructions simultaneously. It incorporates the following six execution units:

- Floating-point unit (FPU)
- Branch processing unit (BPU)
- System register unit (SRU)
- Load/store unit (LSU)
- Two integer units (IUs): IU1 executes all integer instructions. IU2 executes all integer instructions except multiply and divide instructions.

The ability to execute several instructions in parallel and the use of simple instructions with rapid execution times yield high efficiency and throughput for Gekko-based systems. Most integer instructions execute in one clock cycle. The FPU is pipelined, it breaks the tasks it performs into subtasks, and then executes in three successive stages. Typically, a floating-point instruction can occupy only one of the three stages at a time, freeing the previous stage to work on the next floating-point instruction. Thus, three single- or paired single-precision floating-point instructions can be in the FPU execute stage at a time. Double-precision add instructions have a three-cycle latency; double-precision multiply and multiply-add instructions have a four-cycle latency.

Figure 1-1 on Page 1-3 shows the parallel organization of the execution units (shaded in the diagram). The instruction unit fetches, dispatches, and predicts branch instructions. Note that this is a conceptual model that shows basic features rather than attempting to show how features are implemented physically.

Gekko has independent on-chip, 32 Kbyte, eight-way set-associative, physically addressed caches for instructions and data and independent instruction and data memory management units (MMUs). The data cache can be configured as a four-way 16 KByte locked cache and a four-way 16 KByte normal cache. Each MMU has a 128-entry, two-way set-associative translation lookaside buffer (DTLB and ITLB) that saves recently used page address translations. Block address translation is done through the four-entry instruction and data block address translation (IBAT and DBAT) arrays, defined by the PowerPC architecture. During block translation, effective addresses are compared simultaneously with all four BAT entries.

For information about the L1 cache, see Chapter 3, "Gekko Instruction and Data Cache Operation" in this manual.

The L2 cache is implemented with an on-chip, two-way set-associative tag memory, and an on-chip 256 Kbyte SRAM with ECC for data storage.

Gekko has a direct memory access (DMA) engine to transfer data from the external memory to the locked data cache and to transfer data from the locked data cache to the external memory.

A write gather pipe is implemented for effecient non-cacheable store operations.

Gekko has a 32-bit address bus and a 64-bit data bus. Multiple devices compete for system resources through a central external arbiter. Gekko's three-state cache-coherency protocol (MEI) supports the modified, exclusive and invalid states, a compatible subset of the MESI (modified/exclusive/shared/invalid) four-state protocol, and it operates coherently in systems with four-state caches. Gekko supports single-beat and burst data transfers for external memory accesses and memory-mapped I/O operations. The system interface is described in Chapter 7, "Signal Descriptions" and Chapter 8, "Bus Interface Operation" in this manual.

Gekko has four software-controllable power-saving modes. Three static modes, doze, nap, and sleep, progressively reduce power dissipation. When functional units are idle, a dynamic power management mode causes those units to enter a low-power mode automatically without affecting operational performance, software execution, or external hardware. Gekko also provides a thermal assist unit (TAU) and a way to reduce the instruction fetch rate for limiting power dissipation. Power management is described in Chapter 10, "Power and Thermal Management" in this manual.

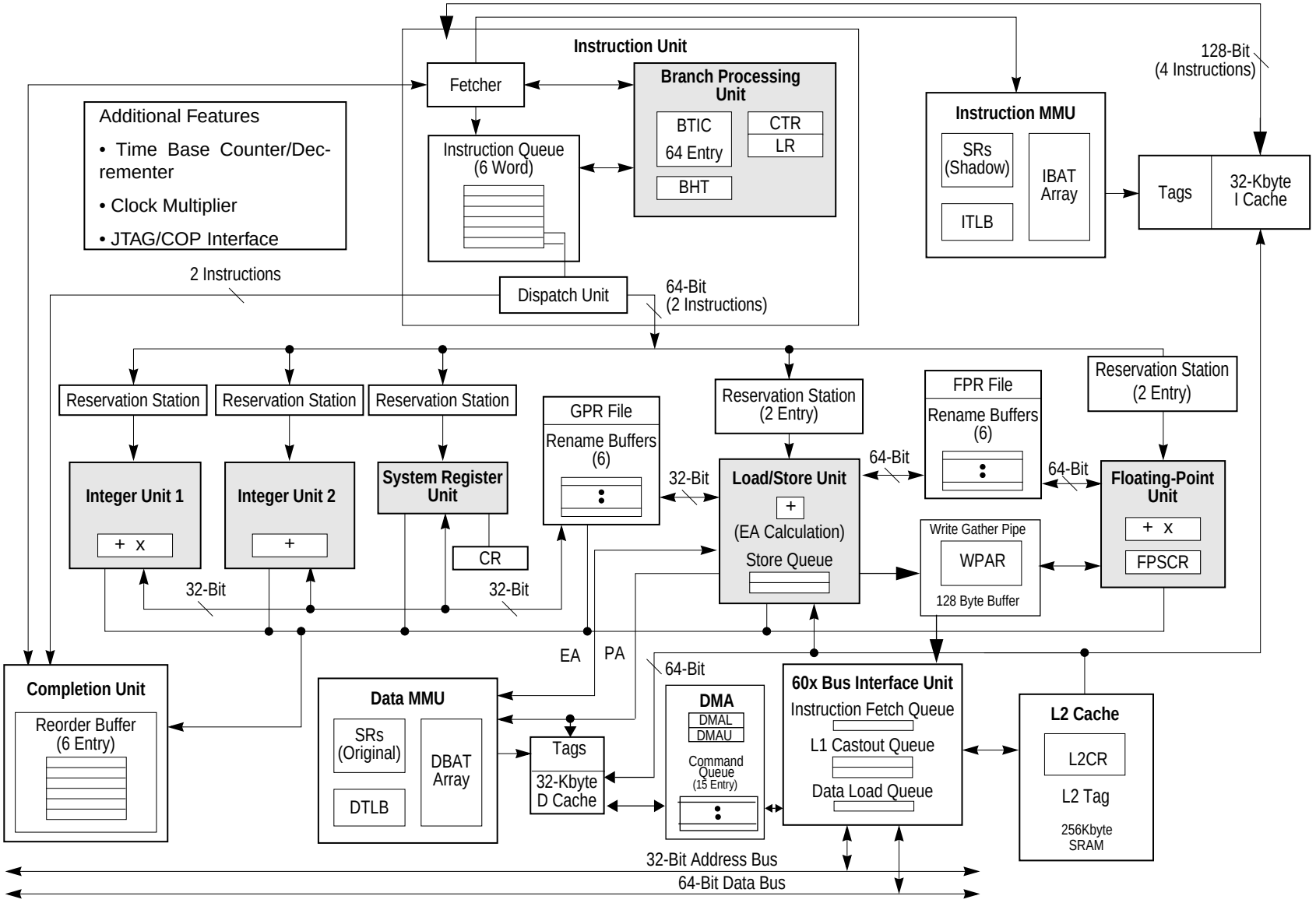


Figure 1-1. Gekko Microprocessor Block Diagram

1.2 Gekko Microprocessor Features

This section lists features of Gekko. The interrelationship of these features is shown in Figure 1-1 on Page 1-3.

1.2.1 Overview of Gekko Microprocessor Features

Major features of Gekko are:

- High-performance, superscalar microprocessor
  - — As many as four instructions can be fetched from the instruction cache per clock cycle
  - — As many as two instructions can be dispatched per clock
  - — As many as six instructions can execute per clock (including two integer instructions)
  - — Single-clock-cycle execution for most instructions
- Six independent execution units and two register files
  - — BPU featuring both static and dynamic branch prediction
    - – 64-entry (16-set, four-way set-associative) branch target instruction cache (BTIC), a cache of branch instructions that have been encountered in branch/loop code sequences. If a target instruction is in the BTIC, it is fetched into the instruction queue a cycle sooner than it can be made available from the instruction cache. Typically, if a fetch access hits the BTIC, it provides the first two instructions in the target stream.
    - – 512-entry branch history table (BHT) with two bits per entry for four levels of prediction—not-taken, strongly not-taken, taken, strongly taken
    - – Branch instructions that do not update the count register (CTR) or link register (LR) are removed from the instruction stream.
  - — Two integer units (IUs) that share thirty-two GPRs for integer operands
    - – IU1 can execute any integer instruction
    - – IU2 can execute all integer instructions except multiply and divide instructions (multiply, divide, shift, rotate, arithmetic, and logical instructions). Most instructions that execute in the IU2 take one cycle to execute. The IU2 has a single-entry reservation station
  - — Three-stage FPU
    - – Fully IEEE 754-1985-compliant FPU for both single- and double-precision operations
    - – Supports paired single-precision floating point arithematic instruction set extension
    - – Supports non-IEEE mode for time-critical operations
    - – Hardware support for denormalized numbers
    - – Two-entry reservation station
    - – Thirty-two 64-bit FPRs for single-, paired single- or double-precision operands.
  - — Two-stage LSU
    - – Two-entry reservation station
    - – Single-cycle, pipelined cache access
    - – Dedicated adder performs EA calculations
    - – Performs alignment and precision conversion for floating-point data
    - – Performs alignment and sign extension for integer data
    - – Three-entry store queue
    - – Supports both big- and little-endian modes
    - – Supports data type conversion with indexed scaling.

  - SRU handles miscellaneous instructions
    - Executes CR logical and Move to/Move from SPR instructions (**mtspr** and **mfspr**)
    - Single-entry reservation station
- Rename buffers
  - Six GPR rename buffers
  - Six FPR rename buffers
  - Condition register buffering supports two CR writes per clock
- Completion unit
  - The completion unit retires an instruction from the six-entry reorder buffer (completion queue) when all instructions ahead of it have been completed, the instruction has finished execution, and no exceptions are pending.
  - Guarantees sequential programming model (precise exception model)
  - Monitors all dispatched instructions and retires them in order
  - Tracks unresolved branches and flushes instructions from the mispredicted branch
  - Retires as many as two instructions per clock
- Separate on-chip instruction and data caches (Harvard architecture)
  - 32-Kbyte, eight-way set-associative instruction and data caches
  - Pseudo least-recently-used (PLRU) replacement algorithm
  - 32-byte (eight-word) cache block
  - Physically indexed/physical tags. (Note that the PowerPC architecture refers to physical address space as real address space.)
  - Cache write-back or write-through operation programmable on a per-page or per-block basis
  - Instruction cache can provide four instructions per clock; data cache can provide two words per clock
  - Caches can be disabled in software
  - Caches can be locked in software
  - Data cache coherency (MEI) maintained in hardware
  - The critical double word is made available to the requesting unit when it is burst into the line-fill buffer. The cache is nonblocking, so it can be accessed during this operation.
  - Data cache can be partitioned as a four-way 16 Kbyte normal cache and a four-way 16-Kbyte locked cache.
- On-chip 1:1 L2 cache.
  - 256 Kbyte on-chip ECC SRAMs
  - On-chip 2-way set-associative tag memory
- DMA engine.
  - 15 entry DMA command queue.
  - Each DMA command can transfer up to 4 Kbyte data in 32 byte increment.
- Write gather pipe.
  - 128 byte circular FIFO buffer.
  - Non-cacheable stores to a specified address are gathered for burst transaction transfer.
- Separate memory management units (MMUs) for instructions and data
  - 52-bit virtual address; 32-bit physical address
  - Address translation for 4-Kbyte pages, variable-sized blocks, and 256-Mbyte segments

  - Memory programmable as write-back/write-through, cacheable/noncacheable, and coherency enforced/coherency not enforced on a page or block basis
  - Separate IBATs and DBATs (four each) also defined as SPRs
  - Separate instruction and data translation lookaside buffers (TLBs)
    - Both TLBs are 128-entry, two-way set associative, and use LRU replacement algorithm
    - TLBs are hardware-reloadable (that is, the page table search is performed in hardware.
- Bus interface features include the following
    - Selectable bus-to-core clock frequency ratios of 2x, 2.5x, 3x, 3.5x, 4x, 4.5x ... 8x and 10x. (2x to 8x, all half-clock multipliers in-between)
    - A 64-bit, split-transaction external data bus with burst transfers
    - Support for address pipelining and limited out-of-order bus transactions
    - Single-entry load queue
    - Single-entry instruction fetch queue
    - Two-entry L1 cache castout queue
    - No-$\overline{\text{DRTRY}}$ mode eliminates the $\overline{\text{DRTRY}}$ signal from the qualified bus grant. This allows the forwarding of data during load operations to the internal core one bus cycle sooner than if the use of $\overline{\text{DRTRY}}$ is enabled.
- Multiprocessing support features include the following:
  - Hardware-enforced, three-state cache coherency protocol (MEI) for data cache.
  - Load/store with reservation instruction pair for atomic memory references, semaphores, and other multiprocessor operations
- Power and thermal management
  - Three static modes, doze, nap, and sleep, progressively reduce power dissipation:
    - Doze—All the functional units are disabled except for the time base/decrementer registers and the bus snooping logic.
    - Nap—The nap mode further reduces power consumption by disabling bus snooping, leaving only the time base register and the PLL in a powered state.
    - Sleep—All internal functional units are disabled, after which external system logic may disable the PLL and SYSCLK.
  - Thermal management facility provides software-controllable thermal management. Thermal management is performed through the use of three supervisor-level registers and an Gekko-specific thermal management exception.
  - Instruction cache throttling provides control of instruction fetching to limit power consumption.
- Performance monitor can be used to help debug system designs and improve software efficiency.
- In-system testability and debugging features through JTAG boundary-scan capability

1.2.2 Instruction Flow

As shown in Figure 1-1 on Page 1-3, the Gekko instruction unit provides centralized control of instruction flow to the execution units. The instruction unit contains a sequential fetcher, six-entry instruction queue (IQ), dispatch unit, and BPU. It determines the address of the next instruction to be fetched based on information from the sequential fetcher and from the BPU.

See Chapter 6, "Instruction Timing" in this manual for a detailed discussion of instruction timing.

The sequential fetcher loads instructions from the instruction cache into the instruction queue. The

BPU extracts branch instructions from the sequential fetcher. Branch instructions that cannot be resolved immediately are predicted using either Gekko-specific dynamic branch prediction or the architecture-defined static branch prediction.

Branch instructions that do not affect the LR or CTR are removed from the instruction stream. The BPU folds branch instructions when a branch is taken (or predicted as taken); branch instructions that are not taken, or predicted as not taken, are removed from the instruction stream through the dispatch mechanism.

Instructions issued beyond a predicted branch do not complete execution until the branch is resolved, preserving the programming model of sequential execution. If branch prediction is incorrect, the instruction unit flushes all predicted path instructions, and instructions are fetched from the correct path.

1.2.2.1 Instruction Queue and Dispatch Unit

The instruction queue (IQ), shown inFigure 1-1 on Page 1-3, holds as many as six instructions and loads up to four instructions from the instruction cache during a single processor clock cycle. The instruction fetcher continuously attempts to load as many instructions as there were vacancies in the IQ in the previous clock cycle. All instructions except branch instructions are dispatched to their respective execution units from the bottom two positions in the instruction queue (IQ0 and IQ1) at a maximum rate of two instructions per cycle. Reservation stations are provided for the IU1, IU2, FPU, LSU, and SRU. The dispatch unit checks for source and destination register dependencies, determines whether a position is available in the completion queue, and inhibits subsequent instruction dispatching as required.

Branch instructions can be detected, decoded, and predicted from anywhere in the instruction queue. For a more detailed discussion of instruction dispatch, see Section 6.6.1 on Page 6-27.

1.2.2.2 Branch Processing Unit (BPU)

The BPU receives branch instructions from the sequential fetcher and performs CR lookahead operations on conditional branches to resolve them early, achieving the effect of a zero-cycle branch in many cases.

Unconditional branch instructions and conditional branch instructions in which the condition is known can be resolved immediately. For unresolved conditional branch instructions, the branch path is predicted using either the architecture-defined static branch prediction or Gekko-specific dynamic branch prediction. Dynamic branch prediction is enabled if HID0[BHT] = 1.

When a prediction is made, instruction fetching, dispatching, and execution continue from the predicted path, but instructions can not complete and write back results to architected registers until the prediction is determined to be correct (resolved).

When a prediction is incorrect, the instructions from the incorrect path are flushed from the processor and processing begins from the correct path.

Gekko allows a second branch instruction to be predicted; instructions from the second predicted instruction stream can be fetched but cannot be dispatched.

Dynamic prediction is implemented using a 512-entry branch history table (BHT), a cache that provides two bits per entry that together indicate four levels of prediction for a branch instruction—not-taken, strongly not-taken, taken, strongly taken. When dynamic branch prediction is disabled, the BPU uses a bit in the instruction encoding to predict the direction of the conditional branch. Therefore, when an unresolved conditional branch instruction is encountered, Gekko executes instructions from the predicted target stream although the results are not committed to architected registers until the conditional branch is resolved. This execution can continue until a second unresolved branch instruction is encountered.

When a branch is taken (or predicted as taken), the instructions from the untaken path must be flushed and the target instruction stream must be fetched into the IQ. The BTIC is a 64-entry cache that contains the most recently used branch target instructions, typically in pairs. When an instruction fetch hits in the BTIC, the instructions arrive in the instruction queue in the next clock cycle, a clock cycle sooner than they would arrive from the instruction cache. Additional instructions arrive from the instruction cache in the next clock cycle. The BTIC reduces the number of missed opportunities to dispatch instructions and gives the processor a one-cycle head start on processing the target stream.

The BPU contains an adder to compute branch target addresses and three user-control registers—the link register (LR), the count register (CTR), and the CR. The BPU calculates the return pointer for subroutine calls and saves it into the LR for certain types of branch instructions. The LR also contains the branch target address for the Branch Conditional to Link Register (**bclr***x*) instruction. The CTR contains the branch target address for the Branch Conditional to Count Register (**bcctr***x*) instruction. Because the LR and CTR are SPRs, their contents can be copied to or from any GPR. Because the BPU uses dedicated registers rather than GPRs or FPRs, execution of branch instructions is largely independent from execution of integer and floating-point instructions.

1.2.2.3 Completion Unit

The completion unit operates closely with the instruction unit. Instructions are fetched and dispatched in program order. At the point of dispatch, the program order is maintained by assigning each dispatched instruction a successive entry in the six-entry completion queue. The completion unit tracks instructions from dispatch through execution and retires them in program order from the two bottom entries in the completion queue (CQ0 and CQ1).

Instructions cannot be dispatched to an execution unit unless there is a vacancy in the completion queue. Branch instructions that do not update the CTR or LR are removed from the instruction stream and do not take an entry in the completion queue. Instructions that update the CTR and LR follow the same dispatch and completion procedures as non-branch instructions, except that they are not issued to an execution unit.

Completing an instruction commits execution results to architected registers (GPRs, FPRs, LR, and CTR). In-order completion ensures the correct architectural state when Gekko must recover from a mispredicted branch or any exception. Retiring an instruction removes it from the completion queue.

For a more detailed discussion of instruction completion, see Section 6.6.1 on Page 6-27.

1.2.2.4 Independent Execution Units

In addition to the BPU, Gekko has five execution units:

- Two Integer Units (IUs)
- A Floating-Point Unit (FPU)
- A Load/Store Unit (LSU)
- A System Register Unit (SRU)

Each is described in the following sections.

1.2.2.4.1 Integer Units (IUs)

The integer units IU1 and IU2 are shown in Figure 1-1 on Page 1-3. The IU1 can execute any integer instruction; the IU2 can execute any integer instruction except multiplication and division instructions. Each IU has a single-entry reservation station that can receive instructions from the dispatch unit and operands from the GPRs or the rename buffers.

Each IU consists of three single-cycle subunits—a fast adder/comparator, a subunit for logical operations, and a subunit for performing rotates, shifts, and count-leading-zero operations. These subunits handle all one-cycle arithmetic instructions; only one subunit can execute an instruction at a time.

The IU1 has a 32-bit integer multiplier/divider as well as the adder, shift, and logical units of the IU2. The multiplier supports early exit for operations that do not require full 32- x 32-bit multiplication.

Each IU has a dedicated result bus (not shown in Figure 1-1 on Page 1-3) that connects to rename buffers.

1.2.2.4.2 Floating-Point Unit (FPU)

The FPU, shown in Figure 1-1 on Page 1-3, is designed such that single- or paired single-precision operations require only a single pass, with a latency of three cycles. As instructions are dispatched to the FPU's reservation station, source operand data can be accessed from the FPRs or from the FPR rename buffers. Results in turn are written to the rename buffers and are made available to subsequent instructions. Instructions pass through the reservation station in dispatch order. The FPU contains two single-precision multiply-add arrays and the floating-point status and control register (FPSCR). The multiply-add array allows Gekko to efficiently implement multiply and multiply-add operations. The FPU is pipelined such that one single-, paired single- or double-precision instruction can be issued per clock cycle. Thirty-two 64-bit floating-point registers are provided to support floating-point operations. Stalls due to contention for FPRs are minimized by automatic allocation of the six floating-point rename registers. Gekko writes the contents of the rename registers to the appropriate FPR when floating-point instructions are retired by the completion unit.

Gekko supports all IEEE 754 floating-point data types (normalized, denormalized, NaN, zero, and infinity) in hardware, eliminating the latency incurred by software exception routines. (Note that "exception" is also referred to as "interrupt" in the architecture specification.) For paired single-precision operations, both data paths comply with the IEEE standard independently.

1.2.2.4.3 Load/Store Unit (LSU)

The LSU executes all load and store instructions and provides the data transfer interface between the GPRs, FPRs, and the cache/memory subsystem. The LSU calculates effective addresses, performs data alignment, and provides sequencing for load/store string and multiple instructions.

Gekko implements 8 paired single quantization load and store instructions. The load instructions read a pair of 8- or 16-bit, signed or unsigned integers, convert them into single-precision floating

point data with the scaling factor in the quantization register, and write the results into the FPR. The store instructions read the 64-bit data from the FPR as a pair of single-precision floating point data, convert the single-precision floating point numbers into a pair of 8- or 16-bit, signed or unsigned integer data, and store the results.

Load and store instructions are translated and issued in program order; however, some memory accesses can occur out of order. Synchronizing instructions can be used to enforce strict ordering. When there are no data dependencies and the guarded bit for the page or block is cleared, a maximum of one out-of-order cacheable load operation can execute per cycle, with a two-cycle total latency on a cache hit. Data returned from the cache is held in a rename register until the completion logic commits the value to a GPR or FPR. Stores cannot be executed out of order and are held in the store queue until the completion logic signals that the store operation is to be completed to memory. Gekko executes store instructions with a maximum throughput of one per cycle and a three-cycle total latency to the data cache. The time required to perform the actual load or store operation depends on the processor/bus clock ratio and whether the operation involves the on-chip cache, the L2 cache, system memory, or an I/O device.

1.2.2.4.4 System Register Unit (SRU)

The SRU executes various system-level instructions, as well as condition register logical operations and move to/from special-purpose register instructions. To maintain system state, most instructions executed by the SRU are execution-serialized; that is, the instruction is held for execution in the SRU until all previously issued instructions have executed. Results from execution-serialized instructions executed by the SRU are not available or forwarded for subsequent instructions until the instruction completes.

1.2.3 Memory Management Units (MMUs)

Gekko's MMUs support up to 4 Petabytes (2^{52}) of virtual memory and 4 Gigabytes (2^{32}) of physical memory for instructions and data. The MMUs also control access privileges for these spaces on block and page granularities. Referenced and changed status is maintained by the processor for each page to support demand-paged virtual memory systems.

The LSU calculates effective addresses for data loads and stores; the instruction unit calculates effective addresses for instruction fetching. The MMU translates the effective address to determine the correct physical address for the memory access.

Gekko supports the following types of memory translation:

- Real addressing mode—In this mode, translation is disabled by clearing bits in the machine state register (MSR): MSR[IR] for instruction fetching or MSR[DR] for data accesses. When address translation is disabled, the physical address is identical to the effective address.
- Page address translation—translates the page frame address for a 4-Kbyte page size
- Block address translation—translates the base address for blocks (128 Kbytes to 256 Mbytes)

If translation is enabled, the appropriate MMU translates the higher-order bits of the effective address into physical address bits. The lower-order address bits (that are untranslated and therefore, considered both logical and physical) are directed to the on-chip caches where they form the index into the eight-way set-associative tag array. After translating the address, the MMU passes the higher-order physical address bits to the cache and the cache lookup completes. For caching-inhibited accesses or accesses that miss in the cache, the untranslated lower-order address bits are concatenated with the translated higher-order address bits; the resulting 32-bit physical address is used by the memory unit and the system interface, which accesses external memory.

The TLBs store page address translations for recent memory accesses. For each access, an effective address is presented for page and block translation simultaneously. If a translation is found in both

the TLB and the BAT array, the block address translation in the BAT array is used. Usually the translation is in a TLB and the physical address is readily available to the on-chip cache. When a page address translation is not in a TLB, hardware searches for one in the page table following the model defined by the PowerPC architecture.

Instruction and data TLBs provide address translation in parallel with the on-chip cache access, incurring no additional time penalty in the event of a TLB hit. Gekko's TLBs are 128-entry, two-way set-associative caches that contain instruction and data address translations. Gekko automatically generates a TLB search on a TLB miss.

1.2.4 On-Chip Level 1 Instruction and Data Caches

Gekko implements separate instruction and data caches. Each cache is 32-Kbyte and eight-way set associative. As defined by the PowerPC architecture, they are physically indexed. Each cache block contains eight contiguous words from memory that are loaded from an 8-word boundary (that is, bits EA[27–31] are zeros); thus, a cache block never crosses a page boundary. An entire cache block can be updated by a four-beat burst load. Misaligned accesses across a page boundary can incur a performance penalty. Caches are nonblocking, write-back caches with hardware support for reloading on cache misses. The critical double word is transferred on the first beat and is simultaneously written to the cache and forwarded to the requesting unit, minimizing stalls due to load delays. The cache being loaded is not blocked to internal accesses while the load completes.

Gekko cache organization is shown in Figure 1-2.

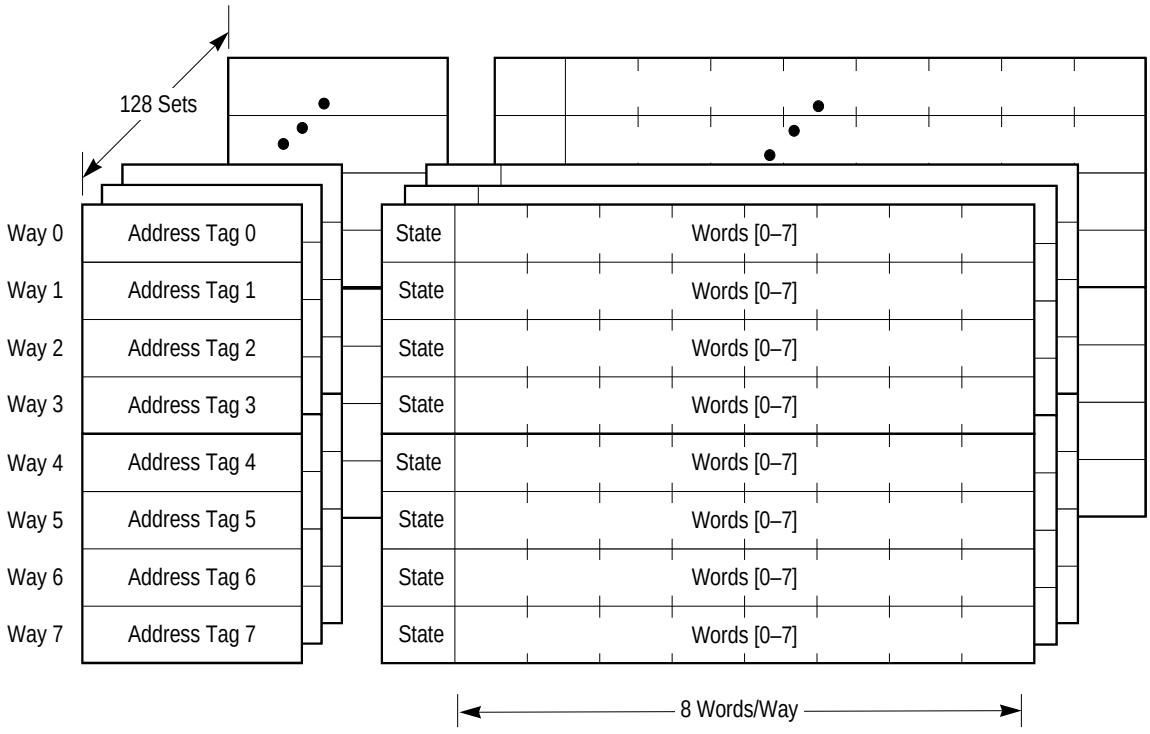


Figure 1-2. Cache Organization

Within one cycle, the data cache provides double-word access to the LSU. Like the instruction cache, the data cache can be invalidated all at once or on a per-cache-block basis. The data cache can be disabled and invalidated by clearing HID0[DCE] and setting HID0[DCFI]. The data cache can be locked by setting HID0[DLOCK]. To ensure cache coherency, the data cache supports the three-state MEI protocol. The data cache tags are single-ported, so a simultaneous load or store and

a snoop access represent a resource collision. If a snoop hit occurs, the LSU is blocked internally for one cycle to allow the eight-word block of data to be copied to the write-back buffer.

By setting HID2[LCE] = 1 , the data cache can be configured into two partitions. The first partition, consisting of ways 0-3, forms a 16 Kbytes normal data cache. The second partition, consisting of ways 4-7, forms a 16 Kbyte locked cache which can be used as an on-chip memory. The detail operation is defined in Chapter 9, "L2 Cache, Locked D-Cache, DMA and Write Gather Pipe" in this manual. Within one cycle, the instruction cache provides up to four instructions to the instruction queue. The instruction cache can be invalidated entirely or on a cache-block basis. The instruction cache can be disabled and invalidated by clearing HID0[ICE] and setting HID0[ICFI]. The instruction cache can be locked by setting HID0[ILOCK]. The instruction cache supports only the valid/invalid states.

Gekko also implements a 64-entry (16-set, four-way set-associative) branch target instruction cache (BTIC). The BTIC is a cache of branch instructions that have been encountered in branch/loop code sequences. If the target instruction is in the BTIC, it is fetched into the instruction queue a cycle sooner than it can be made available from the instruction cache. Typically the BTIC contains the first two instructions in the target stream. The BTIC can be disabled and invalidated through software.

For more information and timing examples showing cache hit and cache miss latencies, see Section 6.3.2 on Page 6-8.

1.2.5 On-Chip Level 2 Cache Implementation

The L2 cache is a unified cache that receives memory requests from both the L1 instruction and data caches independently. The L2 cache is implemented with an on-chip, two-way, set-associative tag memory, and with a 256 Kbyte on-chip SRAM for data storage. The L2 cache normally operates in write-back mode and supports system cache coherency through snooping.

The L2 cache is organized into 64-byte lines, which in turn are subdivided into 32-byte sectors (blocks), the unit at which cache coherency is maintained.

The L2 cache controller contains the L2 cache control register (L2CR) and the L2 cache tag array. The L2CR register includes bits to manage the L2 cache. The cache is two-way set-associative with 2K tags per way. Each sector (32-byte cache block) has its own valid and modified status bits.

Requests from the L1 cache generally result from instruction misses, data load or store misses, write-through operations, or cache management instructions. Requests from the L1 cache are looked up in the L2 tags and serviced by the L2 cache if they hit; they are forwarded to the bus interface if they miss.

The L2 cache can accept multiple, simultaneous accesses. The L1 instruction cache can request an instruction at the same time that the L1 data cache is requesting one load and two store operations. The L2 cache also services snoop requests from the bus. If there are multiple pending requests to the L2 cache, snoop requests have highest priority. The next priority consists of load and store requests from the L1 data cache. The next priority consists of instruction fetch requests from the L1 instruction cache. For more information, see Chapter 9, "L2 Cache, Locked D-Cache, DMA and Write Gather Pipe" in this manual.

1.2.6 System Interface/Bus Interface Unit (BIU)

The address and data buses operate independently; address and data tenures of a memory access are decoupled to provide a more flexible control of memory traffic. The primary activity of the system interface is transferring data and instructions between the processor and system memory. There are two types of memory accesses:

- Single-beat transfers—These memory accesses allow transfer sizes of 8, 16, 24, 32, or 64 bits in one bus clock cycle. Single-beat transactions are caused by uncacheable read and write operations that access memory directly (that is, when caching is disabled), cache-inhibited accesses, and stores in write-through mode.
- Four-beat burst (32 bytes) data transfers—Burst transactions, which always transfer an entire cache block (32 bytes), are initiated when an entire cache block is transferred. Because the first-level caches on Gekko are write-back caches, burst-read memory, burst operations are the most common memory accesses, followed by burst-write memory operations, and single-beat (noncacheable or write-through) memory read and write operations.

Gekko also supports address-only operations, variants of the burst and single-beat operations, (for example, atomic memory operations and global memory operations that are snooped), and address retry activity (for example, when a snooped read access hits a modified block in the cache). The broadcast of some address-only operations is controlled through HID0[ABE]. I/O accesses use the same protocol as memory accesses.

Access to the system interface is granted through an external arbitration mechanism that allows devices to compete for bus mastership. This arbitration mechanism is flexible, allowing Gekko to be integrated into systems that implement various fairness and bus parking procedures to avoid arbitration overhead.

Typically, memory accesses are weakly ordered—sequences of operations, including load/store string and multiple instructions, do not necessarily complete in the order they begin—maximizing the efficiency of the bus without sacrificing data coherency. Gekko allows read operations to go ahead of store operations (except when a dependency exists, or in cases where a noncacheable access is performed), and provides support for a write operation to go ahead of a previously queued read data tenure (for example, letting a snoop push be enveloped between address and data tenures of a read operation). Because Gekko can dynamically optimize run-time ordering of load/store traffic, overall performance is improved.

The system interface is specific for each PowerPC microprocessor implementation.

Gekko signals are grouped as shown in Figure 1-3. Test and control signals provide diagnostics for selected internal circuits.

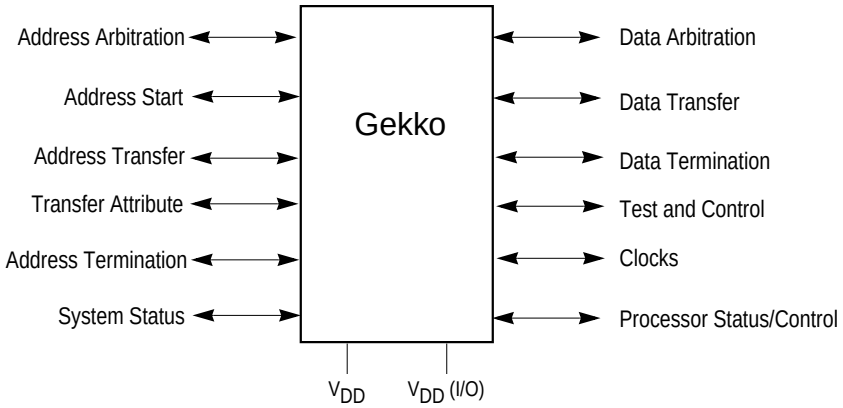


Figure 1-3. System Interface

The system interface supports address pipelining, which allows the address tenure of one transaction to overlap the data tenure of another. The extent of the pipelining depends on external arbitration and control circuitry. Similarly, Gekko supports split-bus transactions for systems with multiple potential bus masters—one device can have mastership of the address bus while another

has mastership of the data bus. Allowing multiple bus transactions to occur simultaneously increases the available bus bandwidth for other activity.

Gekko's clocking structure supports a wide range of processor-to-bus clock ratios.

1.2.7 Signals

Gekko's signals are grouped as follows:

- Address arbitration signals—Gekko uses these signals to arbitrate for address bus mastership.
- Address start signals—These signals indicate that a bus master has begun a transaction on the address bus.
- Address transfer signals—These signals include the address bus and address parity signals. They are used to transfer the address and to ensure the integrity of the transfer.
- Transfer attribute signals—These signals provide information about the type of transfer, such as the transfer size and whether the transaction is bursted, write-through, or caching-inhibited.
- Address termination signals—These signals are used to acknowledge the end of the address phase of the transaction. They also indicate whether a condition exists that requires the address phase to be repeated.
- Data arbitration signals—Gekko uses these signals to arbitrate for data bus mastership.
- Data transfer signals—These signals, which consist of the data bus and data parity signals, are used to transfer the data and to ensure the integrity of the transfer.
- Data termination signals—Data termination signals are required after each data beat in a data transfer. In a single-beat transaction, a data termination signal also indicates the end of the tenure; in burst accesses, data termination signals apply to individual beats and indicate the end of the tenure only after the final data beat. They also indicate whether a condition exists that requires the data phase to be repeated.
- Interrupt signals—These signals include the interrupt signal, checkstop signals, and both soft reset and hard reset signals. These signals are used to generate interrupt exceptions and, under various conditions, to reset the processor.
- Processor status/control signals—These signals are used to set the reservation coherency bit, enable the time base, and other functions.
- Miscellaneous signals—These signals are used in conjunction with such resources as secondary caches and the time base facility.
- JTAG/COP interface signals—The common on-chip processor (COP) unit provides a serial interface to the system for performing board-level boundary scan interconnect tests.
- Clock signals—These signals determine the system clock frequency. These signals can also be used to synchronize multiprocessor systems.

NOTE

A bar over a signal name indicates that the signal is active low—for example, $\overline{\text{ARTRY}}$ (address retry) and $\overline{\text{TS}}$ (transfer start). Active-low signals are referred to as asserted (active) when they are low and negated when they are high. Signals that are not active low, such as AP[0–3] (address bus parity signals) and TT[0–4] (transfer type signals) are referred to as asserted when they are high and negated when they are low.

1.2.8 Signal Configuration

Figure 1-4 shows Gekko's logical pin configuration. The signals are grouped by function.

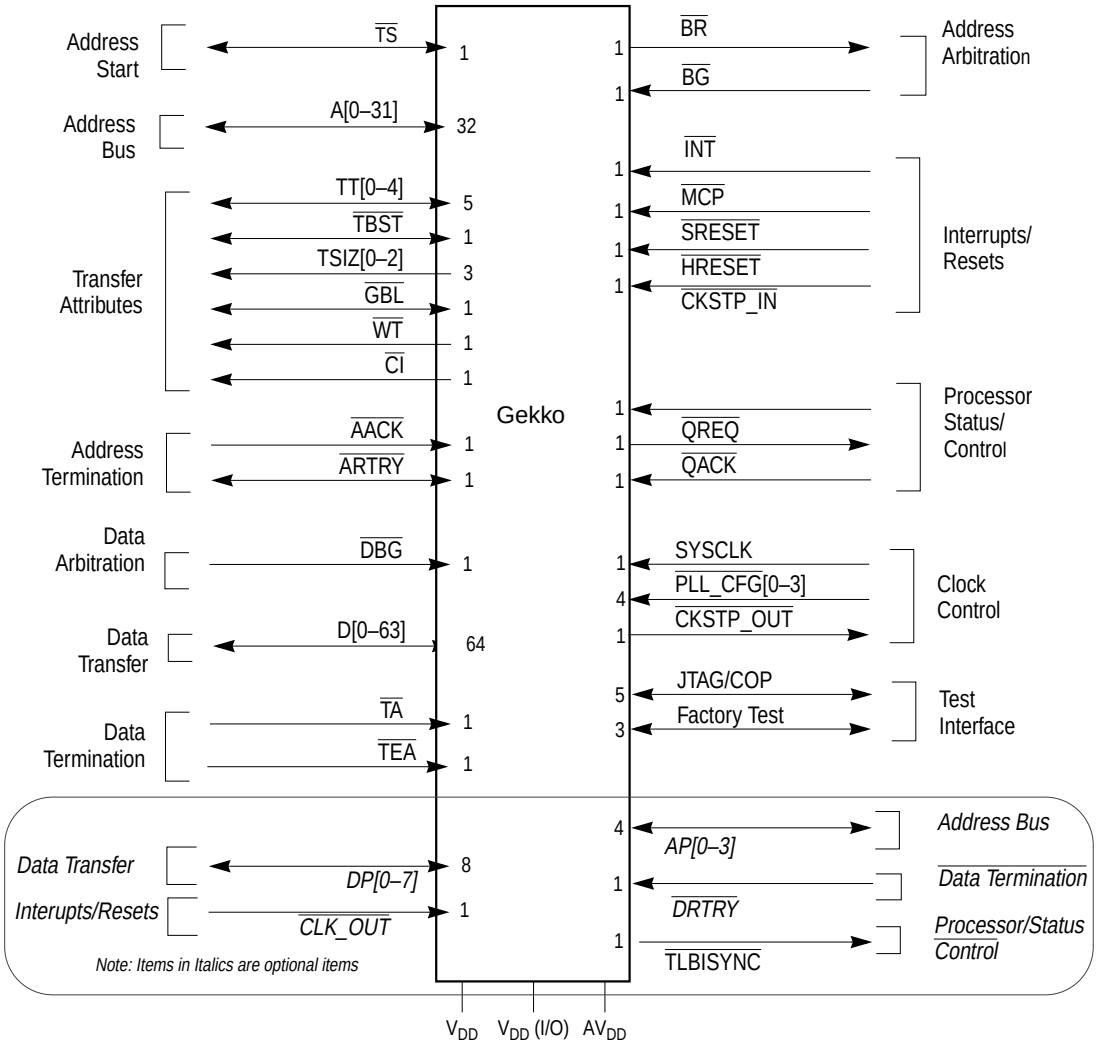


Figure 1-4. Gekko Microprocessor Signal Groups

Signal functionality is described in detail in Chapter 7, "Signal Descriptions" and Chapter 8, "Bus Interface Operation" in this manual.

1.2.9 Clocking

Gekko requires a single system clock input, SYSCLK, that represents the bus interface frequency. Internally, the processor uses a phase-locked loop (PLL) circuit to generate a master core clock that is frequency-multiplied and phase-locked to the SYSCLK input. This core frequency is used to operate the internal circuitry.

The PLL is configured by the PLL_CFG[0–3] signals, which select the multiplier that the PLL uses to multiply the SYSCLK frequency up to the internal core frequency. The feedback in the PLL guarantees that the processor clock is phase locked to the bus clock, regardless of process variations, temperature changes, or parasitic capacitances.

The PLL also ensures a 50% duty cycle for the processor clock.

Gekko supports various processor-to-bus clock frequency ratios, although not all ratios are available for all frequencies. Configuration of the processor/bus clock ratios is displayed through a Gekko-specific register, HID1. For information about supported clock frequencies, see the Gekko hardware specifications.

1.3 Gekko Microprocessor: Implementation

The PowerPC architecture is derived from the POWER architecture (Performance Optimized With Enhanced RISC architecture). The PowerPC architecture shares the benefits of the POWER architecture optimized for single-chip implementations. The PowerPC architecture design facilitates parallel instruction execution and is scalable to take advantage of future technological gains.

This section describes the PowerPC architecture in general, and specific details about the implementation of Gekko as a low-power, 32-bit member of the PowerPC processor family. The structure of this section follows the organization of the user's manual; each subsection provides an overview of each chapter.

- Registers and programming model—Section 1.4 on Page 1-18 describes the registers for the operating environment architecture common among PowerPC processors and describes the programming model. It also describes the registers that are unique to Gekko. The information in this section is described more fully in Chapter 2, "Programming Model" in this manual.
- Instruction set and addressing modes—Section 1.5 on Page 1-23 describes the PowerPC instruction set and addressing modes for the PowerPC operating environment architecture, defines the PowerPC instructions implemented in Gekko, and describes new instruction set extensions to improve the performance of single-precision floating-point operations and the capability of data transfer. The information in this section is described more fully in Chapter 2, "Programming Model" in this manual.
- Cache implementation—Section 1.6 on Page 1-25 describes the cache model that is defined generally for PowerPC processors by the virtual environment architecture. It also provides specific details about Gekko cache implementation. The information in this section is described more fully in Chapter 3, "Gekko Instruction and Data Cache Operation" and Chapter 9, "L2 Cache, Locked D-Cache, DMA and Write Gather Pipe" in this manual.
- Exception model—Section 1.7 on Page 1-25 describes the exception model of the PowerPC operating environment architecture and the differences in Gekko exception model. The information in this section is described more fully in Chapter 4, "Exceptions" in this manual.
- Memory management—Section 1.8 on Page 1-28 describes generally the conventions for memory management among the PowerPC processors. This section also describes Gekko's implementation of the 32-bit PowerPC memory management specification. The information in this section is described more fully in Chapter 5, "Memory Management" in this manual.
- Instruction timing—Section 1.9 on Page 1-29 provides a general description of the instruction timing provided by the superscalar, parallel execution supported by the PowerPC architecture and Gekko. The information in this section is described more fully in Chapter 6, "Instruction Timing" in this manual.
- Power management—Section 1.10 on Page 1-31 describes how the power management can be used to reduce power consumption when the processor, or portions of it, are idle. The information in this section is described more fully in Chapter 10, "Power and Thermal Management" in this manual.

- Thermal management—Section 1.11 on Page 1-32 describes how the thermal management unit and its associated registers (THRM1–THRM3) and exception can be used to manage system activity in a way that prevents exceeding system and junction temperature thresholds. This is particularly useful in high-performance portable systems, which cannot use the same cooling mechanisms (such as fans) that control overheating in desktop systems. The information in this section is described more fully in Chapter 10, "Power and Thermal Management" in this manual.
- Performance monitor—Section 1.12 on Page 1-33 describes the performance monitor facility, which system designers can use to help bring up, debug, and optimize software performance. The information in this section is described more fully in Chapter 11, "Performance Monitor" in this manual.

The following sections summarize the features of Gekko, distinguishing those that are defined by the architecture from those that are unique to Gekko implementation.

The PowerPC architecture consists of the following layers, and adherence to the PowerPC architecture can be described in terms of which of the following levels of the architecture is implemented:

- PowerPC user instruction set architecture (UISA)—Defines the base user-level instruction set, user-level registers, data types, floating-point exception model, memory models for a uniprocessor environment, and programming model for a uniprocessor environment.
- PowerPC virtual environment architecture (VEA)—Describes the memory model for a multiprocessor environment, defines cache control instructions, and describes other aspects of virtual environments. Implementations that conform to the VEA also adhere to the UISA, but may not necessarily adhere to the OEA.
- PowerPC operating environment architecture (OEA)—Defines the memory management model, supervisor-level registers, synchronization requirements, and the exception model. Implementations that conform to the OEA also adhere to the UISA and the VEA.

The PowerPC architecture allows a wide range of designs for such features as cache and system interface implementations. Gekko implementations support the three levels of the architecture described above. For more information about the PowerPC architecture, see the *PowerPC Microprocessor Family: The Programming Environments* manual.

Specific features of Gekko are listed in Section 1.2 on Page 1-4.

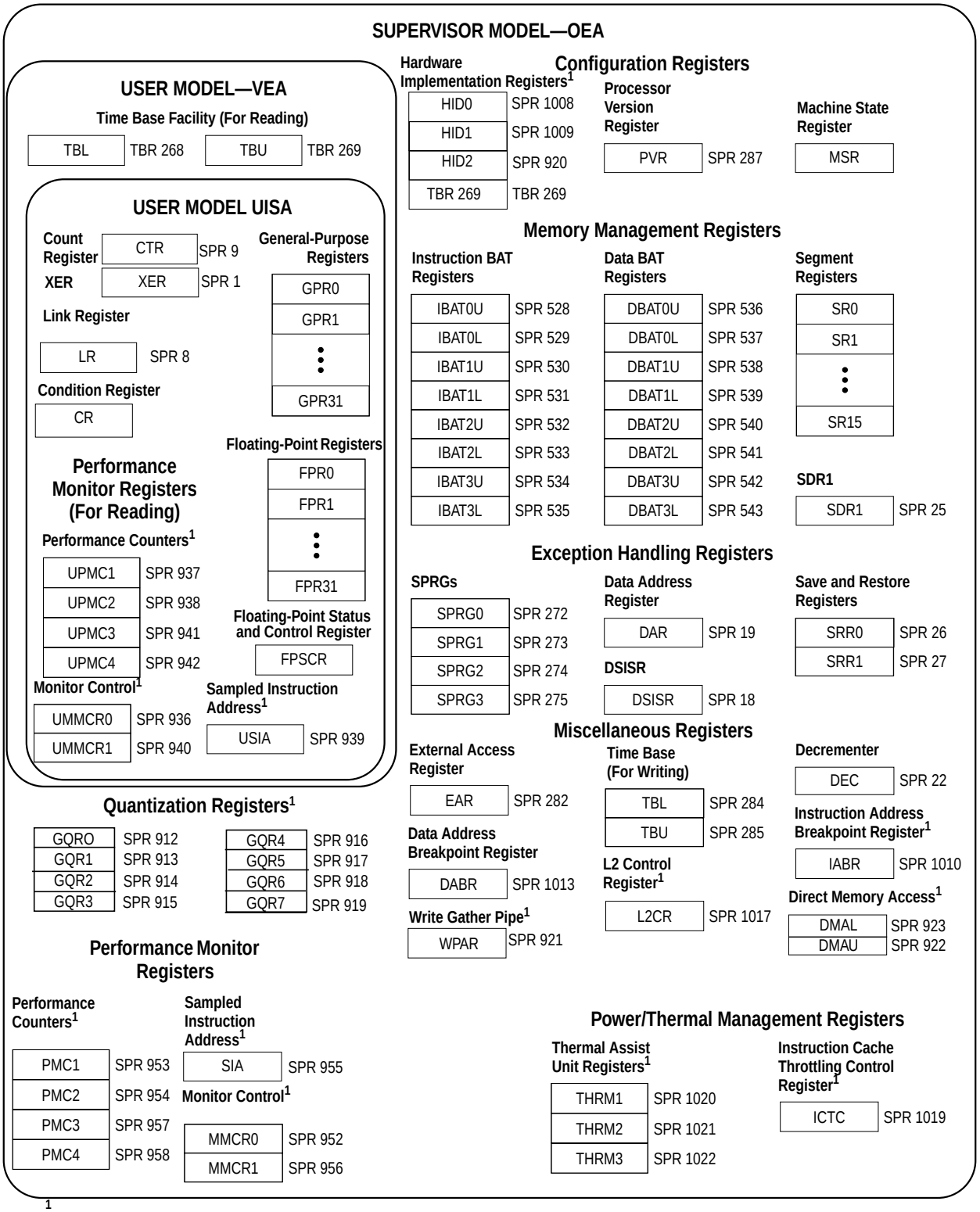
1.4 PowerPC Registers and Programming Model

The PowerPC architecture defines register-to-register operations for most computational instructions. Source operands for these instructions are accessed from the registers or are provided as immediate values embedded in the instruction opcode. The three-register instruction format allows specification of a target register distinct from the two source operands. Load and store instructions transfer data between registers and memory.

PowerPC processors have two levels of privilege—supervisor mode of operation (typically used by the operating system) and user mode of operation (used by the application software). The programming models incorporate 32 GPRs, 32 FPRs, special-purpose registers (SPRs), and several miscellaneous registers. Each PowerPC microprocessor also has its own unique set of hardware implementation-dependent (HID) registers.

Having access to privileged instructions, registers, and other resources allows the operating system to control the application environment (providing virtual memory and protecting operating-system and critical machine resources). Instructions that control the state of the processor, the address translation mechanism, and supervisor registers can be executed only when the processor is operating in supervisor mode.

Figure 1-5 on Page 1-19 shows all Gekko registers available at the user and supervisor level. The numbers to the right of the SPRs indicate the number that is used in the syntax of the instruction operands to access the register. For more information, see Chapter 2, "Programming Model" in this manual.



1 These registers are processor- specific registers. They may not be supported by other PowerPC processors.

Figure 1-5. Gekko Microprocessor Programming Model—Registers

The following tables summarize the PowerPC registers implemented in Gekko; Table 1-1 describes registers (excluding SPRs) defined by the architecture.

Table 1-1. Architecture-Defined Registers (Excluding SPRs)

Register	Level	Function
CR	User	The condition register (CR) consists of eight four-bit fields that reflect the results of certain operations, such as move, integer and floating-point compare, arithmetic, and logical instructions, and provide a mechanism for testing and branching.
FPRs	User	The 32 floating-point registers (FPRs) serve as the data source or destination for floating-point instructions. These 64-bit registers can hold single-, paired single- or double-precision floating-point values.
FPSCR	User	The floating-point status and control register (FPSCR) contains the floating-point exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE-754 standard.
GPRs	User	The 32 GPRs serve as the data source or destination for integer instructions.
MSR	Supervisor	The machine state register (MSR) defines the processor state. Its contents are saved when an exception is taken and restored when exception handling completes. Gekko implements MSR[POW], (defined by the architecture as optional), which is used to enable the power management feature. Gekko-specific MSR[PM] bit is used to mark a process for the performance monitor.
SR0–SR 15	Supervisor	The sixteen 32-bit segment registers (SRs) define the 4-Gbyte space as sixteen 256-Mbyte segments. Gekko implements segment registers as two arrays—a main array for data accesses and a shadow array for instruction accesses; see Figure 1-1 on Page 1-3. Loading a segment entry with the Move to Segment Register (**mtsr**) instruction loads both arrays. The **mfsr** instruction reads the master register, shown as part of the data MMU in Figure 1-1 on Page 1-3.

The OEA defines numerous special-purpose registers that serve a variety of functions, such as providing controls, indicating status, configuring the processor, and performing special operations. During normal execution, a program can access the registers, shown in Figure 1-5 on Page 1-19, depending on the program's access privilege (supervisor or user, determined by the privilege-level (PR) bit in the MSR). GPRs and FPRs are accessed through operands that are part of the instructions. Access to registers can be explicit (that is, through the use of specific instructions for that purpose such as Move to Special-Purpose Register (**mtspr**) and Move from Special-Purpose Register (**mfspr**) instructions) or implicit, as the part of the execution of an instruction. Some registers can be accessed both explicitly and implicitly.

In Gekko, all SPRs are 32 bits wide. Table 1-2 on Page 1-21 describes the architecture-defined SPRs implemented by Gekko. In the *PowerPC Microprocessor Family: The Programming Environments* manual, these registers are described in detail, including bit descriptions.

Section 2.1.1 on Page 2-1 describes how these registers are implemented in Gekko. In particular, this section describes which features the PowerPC architecture defines as optional are implemented on Gekko.

Table 1-2. Architecture-Defined SPRs Implemented

Register	Level	Function
LR	User	The link register (LR) can be used to provide the branch target address and to hold the return address after branch and link instructions.
BATs	Supervisor	The architecture defines 16 block address translation registers (BATs), which operate in pairs. There are four pairs of data BATs (DBATs) and four pairs of instruction BATs (IBATs). BATs are used to define and configure blocks of memory.
CTR	User	The count register (CTR) is decremented and tested by branch-and-count instructions.
DABR	Supervisor	The optional data address breakpoint register (DABR) supports the data address breakpoint facility.
DAR	User	The data address register (DAR) holds the address of an access after an alignment or DSI exception.
DEC	Supervisor	The decrementer register (DEC) is a 32-bit decrementing counter that provides a way to schedule decrementer exceptions.
DSISR	User	The DSISR defines the cause of data access and alignment exceptions.
EAR	Supervisor	The external access register (EAR) controls access to the external access facility through the External Control In Word Indexed (**eciwx**) and External Control Out Word Indexed (**ecowx**) instructions.
PVR	Supervisor	The processor version register (PVR) is a read-only register that identifies the processor.
SDR1	Supervisor	SDR1 specifies the page table format used in virtual-to-physical page address translation.
SRR0	Supervisor	The machine status save/restore register 0 (SRR0) saves the address used for restarting an interrupted program when a Return from Interrupt (**rfi**) instruction executes.
SRR1	Supervisor	The machine status save/restore register 1 (SRR1) is used to save machine status on exceptions and to restore machine status when an **rfi** instruction is executed.
SPRG0–SPRG3	Supervisor	SPRG0–SPRG3 are provided for operating system use.
TB	User: read Supervisor: read/write	The time base register (TB) is a 64-bit register that maintains the time of day and operates interval timers. The TB consists of two 32-bit fields—time base upper (TBU) and time base lower (TBL).
XER	User	The XER contains the summary overflow bit, integer carry bit, overflow bit, and a field specifying the number of bytes to be transferred by a Load String Word Indexed (**lswx**) or Store String Word Indexed (**stswx**) instruction.

Table 1-3 describes the SPRs in Gekko that are not defined by the PowerPC architecture. Section 2.1.2 on Page 2-8 gives detailed descriptions of these registers, including bit descriptions.

Table 1-3. Implementation-Specific Registers

Register	Level	Function
DMAL, DMAU	Supervisor	The DMA upper(DMAU) and DMA low (DMAL) registers are used to issue the DMA commands.
GQR0-GQR7	Supervisor	The quantization registers (GQR0-GQR7) are used to determine the scaling factor and data type conversion for the quantization load/store instructions.
HID0	Supervisor	The hardware implementation-dependent register 0 (HID0) provides checkstop enables and other functions.
HID1	Supervisor	The hardware implementation-dependent register 1 (HID1) allows software to read the configuration of the PLL configuration signals.
HID2	Supervisor	The hardware implementation-dependent register 2 (HID2) enables the paired-single floating-point operations, L1 cache partition, write pipe and DMA, and controls the exceptions associated with the DMA and the locked cache operations..
IABR	Supervisor	The instruction address breakpoint register (IABR) supports instruction address breakpoint exceptions. It can hold an address to compare with instruction addresses in the IQ. An address match causes an instruction address breakpoint exception.
ICTC	Supervisor	The instruction cache-throttling control register (ICTC) has bits for controlling the interval at which instructions are fetched into the instruction buffer in the instruction unit. This helps control Gekko's overall junction temperature.
L2CR	Supervisor	The L2 cache control register (L2CR) is used to configure and operate the L2 cache.
MMCR0–MMCR1	Supervisor	The monitor mode control registers (MMCR0–MMCR1) are used to enable various performance monitoring interrupt functions. UMMCR0–UMMCR1 provide user-level read access to MMCR0–MMCR1.
PMC1–PMC4	Supervisor	The performance monitor counter registers (PMC1–PMC4) are used to count specified events. UPMC1–UPMC4 provide user-level read access to these registers.
SIA	Supervisor	The sampled instruction address register (SIA) holds the EA of an instruction executing at or around the time the processor signals the performance monitor interrupt condition. The USIA register provides user-level read access to the SIA.
THRM1, THRM2	Supervisor	THRM1 and THRM2 provide a way to compare the junction temperature against two user-provided thresholds. The thermal assist unit (TAU) can be operated so that the thermal sensor output is compared to only one threshold, selected in THRM1 or THRM2.
THRM3	Supervisor	THRM3 is used to enable the TAU and to control the output sample time.
UMMCR0–UMMCR1	User	The user monitor mode control registers (UMMCR0–UMMCR1) provide user-level read access to MMCR0–MMCR1.
UPMC1–UPMC4	User	The user performance monitor counter registers (UPMC1–UPMC4) provide user-level read access to PMC1–PMC4.
USIA	User	The user sampled instruction address register (USIA) provides user-level read access to the SIA register.
WPAR	Supervisor	Write gather pipe address register (WPAR) specifies the address of the non-cacheable stores to be gathered for burst transfer.

1.5 Instruction Set

All PowerPC instructions are encoded as single-word (32-bit) opcodes. Instruction formats are consistent among all instruction types, permitting efficient decoding to occur in parallel with operand accesses. This fixed instruction length and consistent format greatly simplify instruction pipelining.

For more information, see Chapter 2, "Programming Model" in this manual.

1.5.1 PowerPC Instruction Set

The PowerPC instructions are divided into the following categories:

- Integer instructions—These include computational and logical instructions.
  - — Integer arithmetic instructions
  - — Integer compare instructions
  - — Integer logical instructions
  - — Integer rotate and shift instructions
- Floating-point instructions—These include floating-point computational instructions, as well as instructions that affect the FPSCR.
  - — Floating-point arithmetic instructions
  - — Floating-point multiply/add instructions
  - — Floating-point rounding and conversion instructions
  - — Floating-point compare instructions
  - — Floating-point status and control instructions
- Load/store instructions—These include integer and floating-point load and store instructions.
  - — Integer load and store instructions
  - — Integer load and store multiple instructions
  - — Floating-point load and store
  - — Primitives used to construct atomic memory operations (**lwarx** and **stwcx.** instructions)
- Flow control instructions—These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow.
  - — Branch and trap instructions
  - — Condition register logical instructions
- Processor control instructions—These instructions are used for synchronizing memory accesses and management of caches, TLBs, and the segment registers.
  - — Move to/from SPR instructions
  - — Move to/from MSR
  - — Synchronize
  - — Instruction synchronize
  - — Order loads and stores
- Memory control instructions—To provide control of caches, TLBs, and SRs.
  - — Supervisor-level cache management instructions
  - — User-level cache instructions
  - — Segment register manipulation instructions
  - — Translation lookaside buffer management instructions

This grouping does not indicate the execution unit that executes a particular instruction or group of instructions.

Integer instructions operate on byte, half-word, and word operands. Floating-point instructions operate on single-precision (one word) and double-precision (one double word) floating-point operands. The PowerPC architecture uses instructions that are four bytes long and word-aligned. It provides for byte, half-word, and word operand loads and stores between memory and a set of 32 GPRs. It also provides for word and double-word operand loads and stores between memory and a set of 32 floating-point registers (FPRs).

Computational instructions do not modify memory. To use a memory operand in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written back to the target location with distinct instructions.

PowerPC processors follow the program flow when they are in the normal execution state; however, the flow of instructions can be interrupted directly by the execution of an instruction or by an asynchronous event. Either kind of exception may cause one of several components of the system software to be invoked.

Effective address computations for both data and instruction accesses use 32-bit unsigned binary arithmetic. A carry from bit 0 is ignored in 32-bit implementations.

1.5.2 Gekko Microprocessor Instruction Set

In addition to the 32-bit single-precision and the 64-bit double-presicion floating-point operands, Gekko implements a new floating-point operand type: paired single-precision. The paired single operand uses a 64-bit FPR to maintain two 32-bit single precision floating point operands. The PowerPC instruction set is substaintially extended to support the paired single data type.

Gekko instruction set is defined as follows:

- Gekko provides hardware support for all 32-bit PowerPC instructions.
- Gekko implements the following instructions optional to the PowerPC architecture:
  - External Control In Word Indexed (**eciwx**)
  - External Control Out Word Indexed (**ecowx**)
  - Floating Select (**fsel**)
  - Floating Reciprocal Estimate Single-Precision (**fres**). Error < 1/4000.
  - Floating Reciprocal Square Root Estimate (**frsqrte**). Error < 1/4000.
  - Store Floating-Point as Integer Word (**stfiw).**
- Gekko implements the following instruction set extension not included in the PowerPC architecture to support the cache line allocation in the locked cache:
  - Data cache block zero and lock (**dcbz_l**).
- Floating point instructions to support the paired single operand data type.Gekko implements the following instruction set extension not included in the PowerPC architecture to support the paired single data type:
  - Quantization load instructions.
  - Quantization store instructions.
  - Floating point instructions to support the paired single operand data type.

1.6 On-Chip Cache Implementation

The following subsections describe the PowerPC architecture's treatment of cache in general, and Gekko-specific implementation, respectively. A detailed description of Gekko cache implementation is provided in Chapter 3, "Gekko Instruction and Data Cache Operation" in this manual.

1.6.1 PowerPC Cache Model

The PowerPC architecture does not define hardware aspects of cache implementations. For example, PowerPC processors can have unified caches, separate instruction and data caches (Harvard architecture), or no cache at all. PowerPC microprocessors control the following memory access modes on a page or block basis:

- Write-back/write-through mode
- Caching-inhibited mode
- Memory coherency

The caches are physically addressed, and the data cache can operate in either write-back or write-through mode, as specified by the PowerPC architecture.

The PowerPC architecture defines the term 'cache block' as the cacheable unit. The VEA and OEA define cache management instructions that a programmer can use to affect cache contents.

1.6.2 Gekko Microprocessor Cache Implementation

Gekko cache implementation is described in Section 1.2.4 on Page 1-11 and Section 1.2.5 on Page 1-12. The BPU also contains a 64-entry BTIC that provides immediate access to cached target instructions. For more information, see Section 1.2.2.2 on Page 1-7.

1.7 Exception Model

The following sections describe the PowerPC exception model and Gekko implementation.
A detailed description of Gekko exception model is provided in Chapter 4, "Exceptions" in this manual.

1.7.1 PowerPC Exception Model

The PowerPC exception mechanism allows the processor to interrupt the instruction flow to handle certain situations caused by external signals, errors, or unusual conditions arising from the instruction execution. When exceptions occur, information about the state of the processor is saved to certain registers, and the processor begins execution at an address (exception vector) predetermined for each exception. Exception processing occurs in supervisor mode.

Although multiple exception conditions can map to a single exception vector, a more specific condition may be determined by examining a register associated with the exception—for example, the DSISR and the FPSCR. Additionally, some exception conditions can be explicitly enabled or disabled by software.

The PowerPC architecture requires that exceptions be handled in program order; therefore, although a particular implementation may recognize exception conditions out of order, they are handled in order. When an instruction-caused exception is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that are undispatched, are required to complete before the exception is taken, and any exceptions those instructions cause must also be handled first; likewise, asynchronous, precise exceptions are recognized when they occur but are not handled until the instructions currently in the completion queue successfully retire or generate an exception, and the completion queue is emptied.

Unless a catastrophic condition causes a system reset or machine check exception, only one exception is handled at a time. For example, if one instruction encounters multiple exception

conditions, those conditions are handled sequentially. After the exception handler handles an exception, the instruction processing continues until the next exception condition is encountered. Recognizing and handling exception conditions sequentially guarantees that exceptions are recoverable.

When an exception is taken, information about the processor state before the exception was taken is saved in SRR0 and SRR1. Exception handlers must save the information stored in SRR0 and SRR1 early to prevent the program state from being lost due to a system reset and machine check exception or due to an instruction-caused exception in the exception handler, and before enabling external interrupts.

The PowerPC architecture supports four types of exceptions:

- Synchronous, precise—These are caused by instructions. All instruction-caused exceptions are handled precisely; that is, the machine state at the time the exception occurs is known and can be completely restored. This means that (excluding the trap and system call exceptions) the address of the faulting instruction is provided to the exception handler and that neither the faulting instruction nor subsequent instructions in the code stream will complete execution before the exception is taken. Once the exception is processed, execution resumes at the address of the faulting instruction (or at an alternate address provided by the exception handler). When an exception is taken due to a trap or system call instruction, execution resumes at an address provided by the handler.
- Synchronous, imprecise—The PowerPC architecture defines two imprecise floating-point exception modes, recoverable and nonrecoverable. Even though Gekko provides a means to enable the imprecise modes, it implements these modes identically to the precise mode (that is, enabled floating-point exceptions are always precise).
- Asynchronous, maskable—The PowerPC architecture defines external and decrementer interrupts as maskable, asynchronous exceptions. When these exceptions occur, their handling is postponed until the next instruction, and any exceptions associated with that instruction, completes execution. If no instructions are in the execution units, the exception is taken immediately upon determination of the correct restart address (for loading SRR0). As shown in Table 1-4, Gekko implements additional asynchronous, maskable exceptions.
- Asynchronous, nonmaskable—There are two nonmaskable asynchronous exceptions: system reset and the machine check exception. These exceptions may not be recoverable, or may provide a limited degree of recoverability. Exceptions report recoverability through the MSR[RI] bit.

1.7.2 Gekko Microprocessor Exception Implementation

Gekko exception classes described above are shown in Table 1-4. Although exceptions have other characteristics, such as priority and recoverability, Table 1-4 describes categories of exceptions Gekko handles uniquely. Table 1-4 includes no synchronous imprecise exceptions; although the PowerPC architecture supports imprecise handling of floating-point exceptions, Gekko implements these exception modes precisely.

Table 1-4. Gekko Microprocessor Exception Classifications

Synchronous/Asynchronous	Precise/Imprecise	Exception Type
Asynchronous, nonmaskable	Imprecise	Machine check, system reset
Asynchronous, maskable	Precise	External, decrementer, system management, performance monitor, and thermal management interrupts
Synchronous	Precise	Instruction-caused exceptions

Table 1-5 lists Gekko exceptions and conditions that cause them. Exceptions specific to Gekko are indicated.

Table 1-5. Exceptions and Conditions

Exception Type	Vector Offset (hex)	Causing Conditions
Reserved	00000	—
System reset	00100	Assertion of either $\overline{\text{HRESET}}$ or $\overline{\text{SRESET}}$ or at power-on reset
Machine check	00200	Assertion of $\overline{\text{TEA}}$ during a data bus transaction, assertion of $\overline{\text{MCP}}$, an address, data or L2 double bit error, DMA queue overflow, DMA look-up misses locked cache, or **dcbz_l** cache hit. MSR[ME] must be set.
DSI	00300	As specified in the PowerPC architecture. For TLB misses on load, store, or cache operations, a DSI exception occurs if a page fault occurs.
ISI	00400	As defined by the PowerPC architecture.
External interrupt	00500	MSR[EE] = 1 and $\overline{\text{INT}}$ is asserted.
Alignment	00600	• A floating-point load/store, **stmw**, **stwcx**, **lmw**, **lwarx**, **eciwx** or **ecowx** instruction operand is not word-aligned. • A multiple/string load/store operation is attempted in little-endian mode. • The operand of **dcbz** or of **dcbz_l** is in memory that is write-through-required or caching-inhibited or the cache is disabled
Program	00700	As defined by the PowerPC architecture.
Floating-point unavailable	00800	As defined by the PowerPC architecture.
Decrementer	00900	As defined by the PowerPC architecture, when the most significant bit of the DEC register changes from 0 to 1 and MSR[EE] = 1.
Reserved	00A00–00BFF	—
System call	00C00	Execution of the System Call (**sc**) instruction.
Trace	00D00	MSR[SE] = 1 or a branch instruction completes and MSR[BE] = 1. Unlike the architecture definition, **isync** does not cause a trace exception
Reserved	00E00	Gekko does not generate an exception to this vector. Other PowerPC processors may use this vector for floating-point assist exceptions.
Reserved	00E10–00EFF	—
Performance monitor[1]	00F00	The limit specified in a PMC register is reached and MMCR0[ENINT] = 1

Table 1-5. Exceptions and Conditions (Continued)

Exception Type	Vector Offset (hex)	Causing Conditions
Instruction address breakpoint[1]	01300	IABR[0–29] matches EA[0–29] of the next instruction to complete, IABR[TE] matches MSR[IR], and IABR[BE] = 1.
Reserved	01400–016FF	—
Thermal management interrupt[1]	01700	Thermal management is enabled, the junction temperature exceeds the threshold specified in THRM1 or THRM2, and MSR[EE] = 1.
Reserved	01800–02FFF	—

Note:
[1] Gekko-specific

1.8 Memory Management

The following subsections describe the memory management features of the PowerPC architecture, and Gekko implementation, respectively. A detailed description of Gekko MMU implementation is provided in Chapter 5, "Memory Management" in this manual.

1.8.1 PowerPC Memory Management Model

The primary functions of the MMU are to translate logical (effective) addresses to physical addresses for memory accesses and to provide access protection on blocks and pages of memory. There are two types of accesses generated by Gekko that require address translation—instruction accesses, and data accesses to memory generated by load, store, and cache control instructions.

The PowerPC architecture defines different resources for 32- and 64-bit processors; Gekko implements the 32-bit memory management model. The memory-management model provides 4 Gbytes of logical address space accessible to supervisor and user programs with a 4-Kbyte page size and 256-Mbyte segment size. BAT block sizes range from 128 Kbyte to 256 Mbyte and are software selectable. In addition, it defines an interim 52-bit virtual address and hashed page tables for generating 32-bit physical addresses.

The architecture also provides independent four-entry BAT arrays for instructions and data that maintain address translations for blocks of memory. These entries define blocks that can vary from 128 Kbytes to 256 Mbytes. The BAT arrays are maintained by system software.

The PowerPC MMU and exception model support demand-paged virtual memory. Virtual memory management permits execution of programs larger than the size of physical memory; demand-paged implies that individual pages are loaded into physical memory from system memory only when they are first accessed by an executing program.

The hashed page table is a variable-sized data structure that defines the mapping between virtual page numbers and physical page numbers. The page table size is a power of 2, and its starting address is a multiple of its size. The page table contains a number of page table entry groups (PTEGs). A PTEG contains eight page table entries (PTEs) of eight bytes each; therefore, each PTEG is 64 bytes long. PTEG addresses are entry points for table search operations.

Setting MSR[IR] enables instruction address translations and MSR[DR] enables data address translations. If the bit is cleared, the respective effective address is the same as the physical address.

1.8.2 Gekko Microprocessor Memory Management Implementation

Gekko implements separate MMUs for instructions and data. It implements a copy of the segment registers in the instruction MMU; however, read and write accesses (**mfsr** and **mtsr**) are handled through the segment registers implemented as part of the data MMU. Gekko MMU is described in Section 1.2.3 on Page 1-10.

The R (referenced) bit is updated in the PTE in memory (if necessary) during a table search due to a TLB miss. Updates to the changed (C) bit are treated like TLB misses. A complete table search is performed and the entire TLB entry is rewritten to update the C bit.

1.9 Instruction Timing

Gekko is a pipelined, superscalar processor. A pipelined processor is one in which instruction processing is divided into discrete stages, allowing work to be done on different instructions in each stage. For example, after an instruction completes one stage, it can pass on to the next stage leaving the previous stage available to the subsequent instruction. This improves overall instruction throughput.

A superscalar processor is one that issues multiple independent instructions into separate execution units, allowing instructions to execute in parallel. Gekko has six independent execution units, two for integer instructions, and one each for floating-point instructions, branch instructions, load and store instructions, and system register instructions. Having separate GPRs and FPRs allows integer, floating-point calculations, and load and store operations to occur simultaneously without interference. Additionally, rename buffers are provided to allow operations to post execution results for use by subsequent instructions without committing them to the architected FPRs and GPRs.

As shown inFigure 1-6, the common pipeline of Gekko has four stages through which all instructions must pass—fetch, decode/dispatch, execute, and complete/write back. Some instructions occupy multiple stages simultaneously and some individual execution units have additional stages. For example, the floating-point pipeline consists of three stages through which all floating-point instructions must pass.

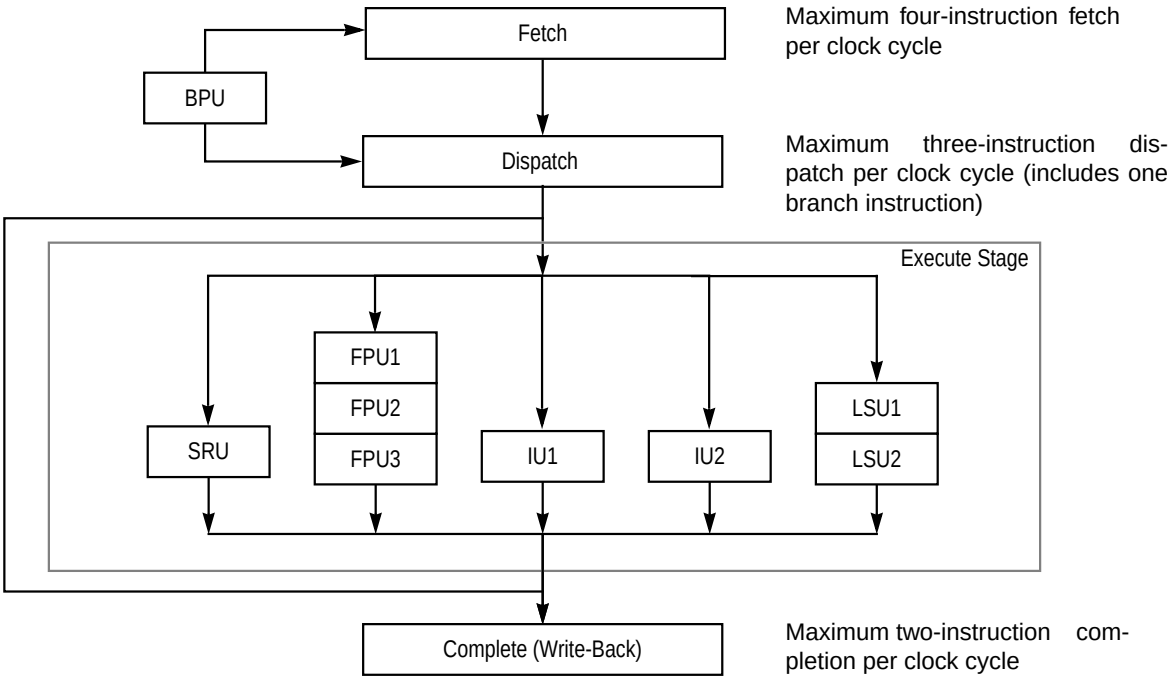


Figure 1-6. Pipeline Diagram

NOTE: Figure 1-6 does not show features, such as reservation stations and rename buffers that reduce stalls and improve instruction throughput.

The instruction pipeline in Gekko has four major pipeline stages, described as follows:

- The fetch pipeline stage primarily involves retrieving instructions from the memory system and determining the location of the next instruction fetch. The BPU decodes branches during the fetch stage and removes those that do not update CTR or LR from the instruction stream.
- The dispatch stage is responsible for decoding the instructions supplied by the instruction fetch stage and determining which instructions can be dispatched in the current cycle. If source operands for the instruction are available, they are read from the appropriate register file or rename register to the execute pipeline stage. If a source operand is not available, dispatch provides a tag that indicates which rename register will supply the operand when it becomes available. At the end of the dispatch stage, the dispatched instructions and their operands are latched by the appropriate execution unit.
- Instructions executed by the IUs, FPU, SRU, and LSU are dispatched from the bottom two positions in the instruction queue. In a single clock cycle, a maximum of two instructions can be dispatched to these execution units in any combination. When an instruction is dispatched, it is assigned a position in the six-entry completion queue. A branch instruction can be issued on the same clock cycle for a maximum three-instruction dispatch.
- During the execute pipeline stage, each execution unit that has an executable instruction executes the selected instruction (perhaps over multiple cycles), writes the instruction's result into the appropriate rename register, and notifies the completion stage that the instruction has finished execution. In the case of an internal exception, the execution unit reports the exception to the completion pipeline stage and (except for the FPU) discontinues instruction execution until the exception is handled. The exception is not signaled until that instruction is

the next to be completed. Execution of most floating-point instructions is pipelined within the FPU allowing up to three instructions to be executing in the FPU concurrently. The FPU stages are multiply, add, and round-convert. Execution of most load/store instructions is also pipelined. The load/store unit has two pipeline stages. The first stage is for effective address calculation and MMU translation and the second stage is for accessing the data in the cache.

- The complete pipeline stage maintains the correct architectural machine state and transfers execution results from the rename registers to the GPRs and FPRs (and CTR and LR, for some instructions) as instructions are retired. As with dispatching instructions from the instruction queue, instructions are retired from the two bottom positions in the completion queue. If completion logic detects an instruction causing an exception, all following instructions are cancelled, their execution results in rename registers are discarded, and instructions are fetched from the appropriate exception vector.

Because the PowerPC architecture can be applied to such a wide variety of implementations, instruction timing varies among PowerPC processors.

For a detailed discussion of instruction timing with examples and a table of latencies for each execution unit, see Chaper 6 “Instruction Timing.”

1.10 Power Management

Gekko provides four power modes, selectable by setting the appropriate control bits in the MSR and HID0 registers. The four power modes are as follows:

- Full-power—This is the default power state of Gekko. Gekko is fully powered and the internal functional units are operating at the full processor clock speed. If the dynamic power management mode is enabled, functional units that are idle will automatically enter a low-power state without affecting performance, software execution, or external hardware.
- Doze—All the functional units of Gekko are disabled except for the time base/decrementer registers and the bus snooping logic. When the processor is in doze mode, an external asynchronous interrupt, a system management interrupt, a decrementer exception, a hard or soft reset, or machine check brings Gekko into the full-power state. Gekko in doze mode maintains the PLL in a fully powered state and locked to the system external clock input (SYSCLK) so a transition to the full-power state takes only a few processor clock cycles.
- Nap—The nap mode further reduces power consumption by disabling bus snooping, leaving only the time base register and the PLL in a powered state. Gekko returns to the full-power state upon receipt of an external asynchronous interrupt, a system management interrupt, a decrementer exception, a hard or soft reset, or a machine check input ($\overline{\text{MCP}}$). A return to full-power state from a nap state takes only a few processor clock cycles. When the processor is in nap mode, if $\overline{\text{QACK}}$ is negated, the processor is put in doze mode to support snooping.
- Sleep—Sleep mode minimizes power consumption by disabling all internal functional units, after which external system logic may disable the PLL and SYSCLK. Returning Gekko to the full-power state requires the enabling of the PLL and SYSCLK, followed by the assertion of an external asynchronous interrupt, a system management interrupt, a hard or soft reset, or a machine check input ($\overline{\text{MCP}}$) signal after the time required to relock the PLL.

Chapter 10, "Power and Thermal Management" in this manual provides information about power saving and thermal management modes for Gekko.

1.11 Thermal Management

Gekko's thermal assist unit (TAU) provides a way to control heat dissipation. This ability is particularly useful in portable computers, which, due to power consumption and size limitations, cannot use desktop cooling solutions such as fans. Therefore, better heat sink designs coupled with intelligent thermal management is of critical importance for high performance portable systems.

Primarily, the thermal management system monitors and regulates the system's operating temperature. For example, if the temperature is about to exceed a set limit, the system can be made to slow down or even suspend operations temporarily in order to lower the temperature.

The thermal management facility also ensures that the processor's junction temperature does not exceed the operating specification. To avoid the inaccuracies that arise from measuring junction temperature with an external thermal sensor, Gekko's on-chip thermal sensor and logic tightly couples the thermal management implementation.

The TAU consists of a thermal sensor, digital-to-analog convertor, comparator, control logic, and the dedicated SPRs described in Section 1.4 on Page 1-18. The TAU does the following:

- Compares the junction temperature against user-programmable thresholds
- Generates a thermal management interrupt if the temperature crosses the threshold
- Enables the user to estimate the junction temperature by way of a software successive approximation routine

The TAU is controlled through the privileged **mtspr**/**mfspr** instructions to the three SPRs provided for configuring and controlling the sensor control logic, which function as follows:

- THRM1 and THRM2 provide the ability to compare the junction temperature against two user-provided thresholds. Having dual thresholds gives the thermal management software finer control of the junction temperature. In single threshold mode, the thermal sensor output is compared to only one threshold in either THRM1 or THRM2.
- THRM3 is used to enable the TAU and to control the comparator output sample time. The thermal management logic manages the thermal management interrupt generation and time multiplexed comparisons in the dual threshold mode as well as other control functions.

Instruction cache throttling provides control of Gekko's overall junction temperature by determining the interval at which instructions are fetched. This feature is accessed through the ICTC register.

Chapter 10, "Power and Thermal Management" in this manual provides information about power saving and thermal management modes for Gekko.

1.12 Performance Monitor

Gekko incorporates a performance monitor facility that system designers can use to help bring up, debug, and optimize software performance. The performance monitor counts events during execution of code, relating to dispatch, execution, completion, and memory accesses.

The performance monitor incorporates several registers that can be read and written to by supervisor-level software. User-level versions of these registers provide read-only access for user-level applications. These registers are described in Section 1.4 on Page 1-18. Performance monitor control registers, MMCR0 or MMCR1, can be used to specify which events are to be counted and the conditions for which a performance monitoring interrupt is taken. Additionally, the sampled instruction address register, SIA (USIA), holds the address of the first instruction to complete after the counter overflowed.

Attempting to write to a user-read-only performance monitor register causes a program exception, regardless of the MSR[PR] setting.

When a performance monitoring interrupt occurs, program execution continues from vector offset 0x00F00.

Chapter 11, "Performance Monitor" in this manual describes the operation of the performance monitor diagnostic tool incorporated in Gekko.

Chapter 2 Programming Model

This chapter describes the Gekko programming model, emphasizing those features specific to the Gekko processor and summarizing those that are common to PowerPC processors. It consists of three major sections, which describe the following:

- Registers implemented in Gekko
- Operand conventions
- The Gekko instruction set

For detailed information about architecture-defined features, see the *PowerPC Microprocessor Family: The Programming Environments* manual.

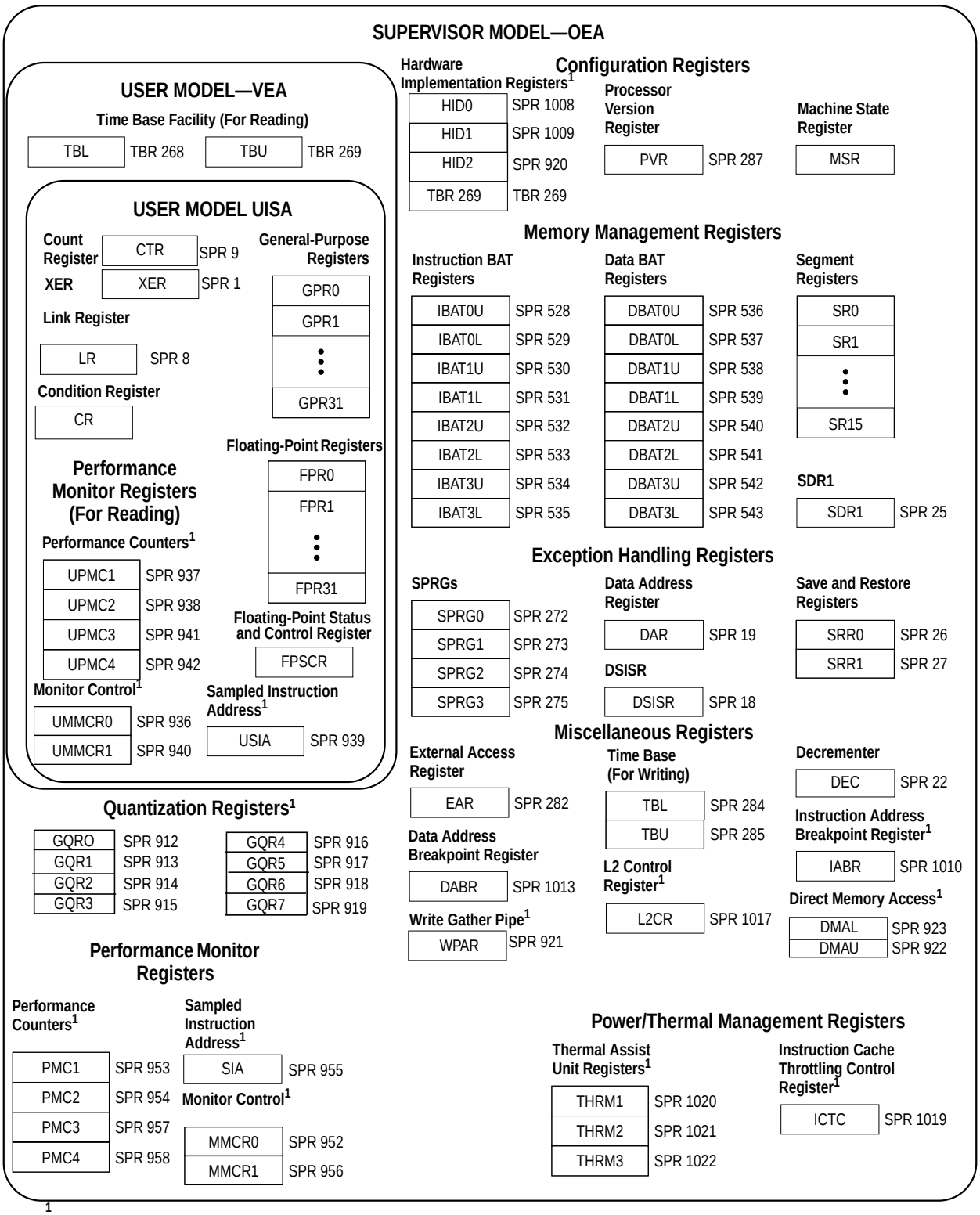
2.1 Gekko Processor Register Set

This section describes the registers implemented in Gekko. It includes an overview of registers defined by the PowerPC architecture, highlighting differences in how these registers are implemented in Gekko, and a detailed description of Gekko-specific registers. Full descriptions of the architecture-defined register set are provided in Chapter 2, “PowerPC Register Set" in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Registers are defined at all three levels of the PowerPC architecture—user instruction set architecture (UISA), virtual environment architecture (VEA), and operating environment architecture (OEA). The PowerPC architecture defines register-to-register operations for all computational instructions. Source data for these instructions are accessed from the on-chip registers or are provided as immediate values embedded in the opcode. The three-register instruction format allows specification of a target register distinct from the two source registers, thus preserving the original data for use by other instructions and reducing the number of instructions required for certain operations. Data is transferred between memory and registers with explicit load and store instructions only.

2.1.1 Register Set

The registers implemented on Gekko are shown in Figure 2-1 on Page 2-2. The number to the right of the special-purpose registers (SPRs) indicates the number that is used in the syntax of the instruction operands to access the register (for example, the number used to access the integer exception register (XER) is SPR 1). These registers can be accessed using the **mtspr** and **mfspr** instructions.



1 These registers are processor-specific registers. They may not be supported by other PowerPC processors.

Figure 2-1. Programming Model—Gekko Microprocessor Registers

The PowerPC UISA registers are user-level. General-purpose registers (GPRs) and floating-point registers (FPRs) are accessed through instruction operands. Access to registers can be explicit (by using instructions for that purpose such as Move to Special-Purpose Register (**mtspr**) and Move from Special-Purpose Register (**mfspr**) instructions) or implicit as part of the execution of an instruction. Some registers are accessed both explicitly and implicitly.

Implementation Note—Gekko fully decodes the SPR field of the instruction. If the SPR specified is undefined, the illegal instruction program exception occurs. The PowerPC's user-level registers are described as follows:

- **User-level registers** (UISA)—The user-level registers can be accessed by all software with either user or supervisor privileges. They include the following:
  - — General-purpose registers (GPRs). The thirty-two GPRs (GPR0–GPR31) serve as data source or destination registers for integer instructions and provide data for generating addresses. See "General Purpose Registers (GPRs)" in Chapter 2, "PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.
  - — Floating-point registers (FPRs). The thirty-two FPRs (FPR0–FPR31) serve as the data source or destination for all floating-point instructions. See "Floating-Point Registers (FPRs)" in Chapter 2, "PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual.
  - — Condition register (CR). The 32-bit CR consists of eight 4-bit fields, CR0–CR7, that reflect results of certain arithmetic operations and provide a mechanism for testing and branching. See "Condition Register (CR)" in Chapter 2, "PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual.
  - — Floating-point status and control register (FPSCR). The FPSCR contains all floating-point exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE 754 standard. See "Floating-Point Status and Control Register (FPSCR)" in Chapter 2, "PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual.

  The remaining user-level registers are SPRs. Note that the PowerPC architecture provides a separate mechanism for accessing SPRs (the **mtspr** and **mfspr** instructions). These instructions are commonly used to explicitly access certain registers, while other SPRs may be more typically accessed as the side effect of executing other instructions.

  - — Integer exception register (XER). The XER indicates overflow and carries for integer operations. See "XER Register (XER)" in Chapter 2, "PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

    Implementation Note—To allow emulation of the **lscbx** instruction defined by the POWER architecture, XER[16–23] is implemented so that they can be read with **mfspr**[XER] and written with **mtxer**[XER] instructions.
  - — Link register (LR). The LR provides the branch target address for the Branch Conditional to Link Register (**bclr***x*) instruction, and can be used to hold the logical address of the instruction that follows a branch and link instruction, typically used for linking to subroutines. See "Link Register (LR)" in Chapter 2, "PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual.

— Count register (CTR). The CTR holds a loop count that can be decremented during execution of appropriately coded branch instructions. The CTR can also provide the branch target address for the Branch Conditional to Count Register (**bcctr***x*) instruction. See “Count Register (CTR)" in Chapter 2, “PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual.

- **User-level registers** (VEA)—The PowerPC VEA defines the time base facility (TB), which consists of two 32-bit registers—time base upper (TBU) and time base lower (TBL). The time base registers can be written to only by supervisor-level instructions but can be read by both user- and supervisor-level software. For more information, see “PowerPC VEA Register Set—Time Base" in Chapter 2, “PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual.
- **Supervisor-level registers** (OEA)—The OEA defines the registers an operating system uses for memory management, configuration, exception handling, and other operating system functions. The OEA defines the following supervisor-level registers for 32-bit implementations:

  — Configuration registers

  – Machine state register (MSR). The MSR defines the state of the processor. The MSR can be modified by the Move to Machine State Register (**mtmsr**), System Call (**sc**), and Return from Exception (**rfi**) instructions. It can be read by the Move from Machine State Register (**mfmsr**) instruction. When an exception is taken, the contents of the MSR are saved to the machine status save/restore register 1 (SRR1), which is described below. See “Machine State Register (MSR)" in Chapter 2, “PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

    Implementation Note—Table 2-1 describes MSR bits Gekko implements that are not required by the PowerPC architecture.

Table 2-1. Additional MSR Bits

Bit	Name	Description
13	POW	Power management enable. Optional to the PowerPC architecture. 0 Power management is disabled. 1 Power management is enabled. The processor can enter a power-saving mode when additional conditions are present. The mode chosen is determined by the DOZE, NAP, and SLEEP bits in the hardware implementation-dependent register 0 (HID0), described in Table 2-4 on Page 2-9.
29	PM	Performance monitor marked mode. This bit is specific to Gekko, and is defined as reserved by the PowerPC architecture. See Chapter 11, "Performance Monitor" in this manual in this manual. 0 Process is not a marked process. 1 Process is a marked process.

NOTE: Setting MSR[EE] masks not only the architecture-defined external interrupt and decrementer exceptions but also the Gekko-specific system management, performance monitor, and thermal management exceptions.

– Processor version register (PVR). This register is a read-only register that identifies the version (model) and revision level of the PowerPC processor. For more information,

see “Processor Version Register (PVR)" in Chapter 2, “PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual.

Implementation Note—The processor version number is 0x7000 for Gekko. Early releases of the hardware may have a processor version number of 0x0008. The processor revision level starts at 0x0100 and is updated for each silicon revision.

— Memory management registers
  – Block-address translation (BAT) registers. The PowerPC OEA includes an array of block address translation registers that can be used to specify four blocks of instruction space and four blocks of data space. The BAT registers are implemented in pairs—four pairs of instruction BATs (IBAT0U–IBAT3U and IBAT0L–IBAT3L) and four pairs of data BATs (DBAT0U–DBAT3U and DBAT0L–DBAT3L). Figure 2-1 on Page 2-2 lists the SPR numbers for the BAT registers. For more information, see “BAT Registers" in Chapter 2, “PowerPC Register Set” of the *PowerPC Microprocessor Family: The Programming Environments* manual. Because BAT upper and lower words are loaded separately, software must ensure that BAT translations are correct during the time that both BAT entries are being loaded.

    Gekko implements the G bit in the IBAT registers; however, attempting to execute code from an IBAT area with G = 1 causes an ISI exception. This complies with the revision of the architecture described in the *PowerPC Microprocessor Family: The Programming Environments* manual.
  – SDR1. The SDR1 register specifies the page table base address used in virtual-to-physical address translation. See “SDR1" in Chapter 2, “PowerPC Register Set” of the *PowerPC Microprocessor Family: The Programming Environments* manual.”
  – Segment registers (SR). The PowerPC OEA defines sixteen 32-bit segment registers (SR0–SR15). Note that the SRs are implemented on 32-bit implementations only. The fields in the segment register are interpreted differently depending on the value of bit 0. See “Segment Registers" in Chapter 2, “PowerPC Register Set” of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

    Note that Gekko implements separate memory management units (MMUs) for instruction and data. It associates the architecture-defined SRs with the data MMU (DMMU). It reflects the values of the SRs in separate, so-called ‘shadow’ segment registers in the instruction MMU (IMMU).

— Exception-handling registers
  – Data address register (DAR). After a DSI or an alignment exception, DAR is set to the effective address (EA) generated by the faulting instruction. See “Data Address Register (DAR)" in Chapter 2, “PowerPC Register Set” of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.
  – SPRG0–SPRG3. The SPRG0–SPRG3 registers are provided for operating system use. See “SPRG0–SPRG3" in Chapter 2, “PowerPC Register Set” of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.
  – DSISR. The DSISR register defines the cause of DSI and alignment exceptions. See

"DSISR" in Chapter 2, "PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

– Machine status save/restore register 0 (SRR0). The SRR0 register is used to save the address of the instruction at which execution continues when **rfi** executes at the end of an exception handler routine. See "Machine Status Save/Restore Register 0 (SRR0)" in Chapter 2, "PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

– Machine status save/restore register 1 (SRR1). The SRR1 register is used to save machine status on exceptions and to restore machine status when **rfi** executes. See "Machine Status Save/Restore Register 1 (SRR1)" in Chapter 2, "PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

Implementation Note—When a machine check exception occurs, Gekko sets one or more error bits in SRR1. Table 2-2 describes SRR1 bits Gekko implements that are not required by the PowerPC architecture.

Table 2-2. Additional SRR1 Bits

Bit	Name	Description
10	DMA	Set by a dcbz_l or DMA error
11	L2DP	Set by a double bit ECC error in the L2.
12	MCPIN	Set by the assertion of $\overline{\text{MCP}}$
13	TEA	Set by a $\overline{\text{TEA}}$ assertion on the 60x bus
14	DP	Set by a data parity error on the 60x bus
15	AP	Set by an address parity error on the 60x bus

— Miscellaneous registers

– Time base (TB). The TB is a 64-bit structure provided for maintaining the time of day and operating interval timers. The TB consists of two 32-bit registers—time base upper (TBU) and time base lower (TBL). The time base registers can be written to only by supervisor-level software, but can be read by both user- and supervisor-level software. See "Time Base Facility (TB)—OEA" in Chapter 2, "PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

– Decrementer register (DEC). This register is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay; the frequency is a subdivision of the processor clock. See "Decrementer Register (DEC)" in Chapter 2, "PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

Implementation Note—In Gekko, the decrementer register is decremented and the time base is incremented at a speed that is one-fourth the speed of the bus clock.

– Data address breakpoint register (DABR)—This optional register is used to cause a breakpoint exception if a specified data address is encountered. See "Data Address Breakpoint Register (DABR)" in Chapter 2, "PowerPC Register Set" of the *PowerPC*

Microprocessor Family: The Programming Environments manual."

  - External access register (EAR). This optional register is used in conjunction with **eciwx** and **ecowx**. Note that the EAR register and the **eciwx** and **ecowx** instructions are optional in the PowerPC architecture and may not be supported in all PowerPC processors that implement the OEA. See "External Access Register (EAR)" in Chapter 2, "PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

- **Gekko-specific registers**—The PowerPC architecture allows implementation-specific SPRs. Those incorporated in Gekko are described as follows. Note that in Gekko, these registers are all supervisor-level registers.
  - Instruction address breakpoint register (IABR)—This register can be used to cause a breakpoint exception if a specified instruction address is encountered.
  - Hardware implementation-dependent register 0 (HID0)—This register controls various functions, such as enabling checkstop conditions, and locking, enabling, and invalidating the instruction and data caches.
  - Hardware implementation-dependent register 1 (HID1)—This register reflects the state of PLL_CFG[0–3] clock signals.
  - Hardware implementation-dependent register 2 (HID2)—This register controls the graphics enhancement facilities, including the locked cache and DMA, the write gather pipe and paired single processing in the floating-point unit.
  - Direct memory access (DMA) registers—The pair of DMA registers, DMAU and DMAL, is used to specify and issue a DMA command. Each DMA command consists of a locked cache address, an external memory address, transfer length and transfer direction.
  - Graphics quantization registers (GQRs)—This array of eight registers is used to specify the conversion parameters used by the paired single quantized load and store instructions.
  - Write pipe address register (WPAR)—This register is used to specify the target address of non-cacheable store transactions to be gathered by the write gather pipe facility.
  - The L2 cache control register (L2CR) is used to configure and operate the L2 cache.
  - Performance monitor registers. The following registers are used to define and count events for use by the performance monitor:
    - The performance monitor counter registers (PMC1–PMC4) are used to record the number of times a certain event has occurred. UPMC1–UPMC4 provide user-level read access to these registers.
    - The monitor mode control registers (MMCR0–MMCR1) are used to enable various performance monitor interrupt functions. UMMCR0–UMMCR1 provide user-level read access to these registers.
    - The sampled instruction address register (SIA) contains the effective address of an instruction executing at or around the time that the processor signals the performance monitor interrupt condition. USIA provides user-level read access to the SIA.
    - Gekko does not implement the sampled data address register (SDA) or the user-level, read-only USDA registers. However, for compatibility with processors that do, those registers can be written to by boot code without causing an exception. SDA is SPR 959; USDA is SPR 943.

— The instruction cache throttling control register (ICTC) has bits for enabling the instruction cache throttling feature and for controlling the interval at which instructions are forwarded to the instruction buffer in the fetch unit. This provides control over the processor's overall junction temperature.
— Thermal management registers (THRM1, THRM2, and THRM3). Used to enable and set thresholds for the thermal management facility.
  – THRM1 and THRM2 provide the ability to compare the junction temperature against two user-provided thresholds. The dual thresholds allow the thermal management software differing degrees of action in lowering the junction temperature. The TAU can be also operated in a single threshold mode in which the thermal sensor output is compared to only one threshold in either THRM1 or THRM2.
  – THRM3 is used to enable the thermal management assist unit (TAU) and to control the comparator output sample time.

Note that while it is not guaranteed that the implementation of Gekko-specific registers is consistent among PowerPC processors, other processors may implement similar or identical registers.

2.1.2 Gekko-Specific Registers

This section describes registers that are defined for Gekko but are not included in the PowerPC architecture.

2.1.2.1 Instruction Address Breakpoint Register (IABR)

The address breakpoint register (IABR), shown in Figure 2-2, supports the instruction address breakpoint exception. When this exception is enabled, instruction fetch addresses are compared with an effective address stored in the IABR. If the word specified in the IABR is fetched, the instruction breakpoint handler is invoked. The instruction that triggers the breakpoint does not execute before the handler is invoked. For more information, see Section 4.5.14, "Instruction Address Breakpoint Exception (0x01300)" on Page 4-21. The IABR can be accessed with **mtspr** and **mfspr** using the SPR1010.

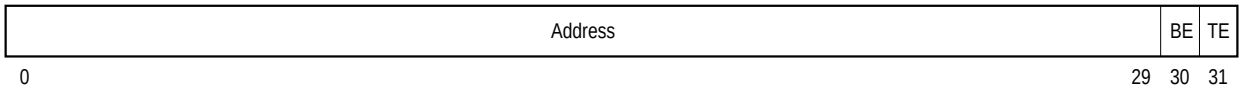


Figure 2-2. Instruction Address Breakpoint Register

The IABR bits are described in Table 2-3.

Table 2-3. Instruction Address Breakpoint Register Bit Settings

Bits	Name	Description
0–29	Address	Word address to be compared
30	BE	Breakpoint enabled. Setting this bit indicates that breakpoint checking is to be done.
31	TE	Translation enabled. An IABR match is signaled if this bit matches MSR[IR].

2.1.2.2 Hardware Implementation-Dependent Register 0

The hardware implementation-dependent register 0 (HID0) controls the state of several functions within Gekko. The HID0 register is shown in Figure 2-3.

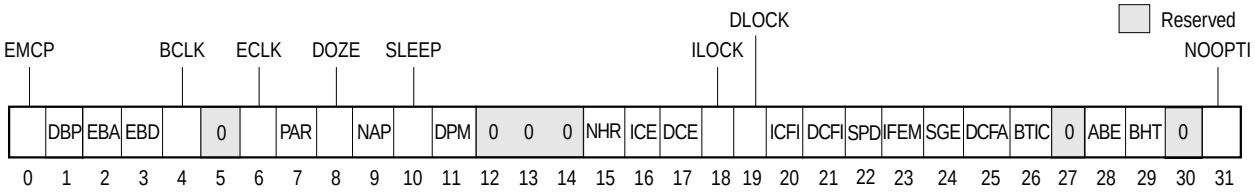


Figure 2-3. Hardware Implementation-Dependent Register 0 (HID0)

The HID0 bits are described in Table 2-4.

Table 2-4. HID0 Bit Functions

Bit	Name	Function
0	EMCP	Enable $\overline{\text{MCP}}$. The primary purpose of this bit is to mask out further machine check exceptions caused by assertion of $\overline{\text{MCP}}$, similar to how MSR[EE] can mask external interrupts. 0 Masks $\overline{\text{MCP}}$. Asserting $\overline{\text{MCP}}$ does not generate a machine check exception or a checkstop. 1 Asserting $\overline{\text{MCP}}$ causes checkstop if MSR[ME] = 0 or a machine check exception if ME = 1.
1	DBP	Disable 60x bus address and data parity generation. 0 Parity generation is enabled. 1 Disable parity generation. If the system does not use address or data parity and the respective parity checking is disabled (HID0[EBA] or HID0[EBD] = 0), input receivers for those signals are disabled, require no pull-up resistors, and thus should be left unconnected. If all parity generation is disabled, all parity checking should also be disabled and parity signals need not be connected.
2	EBA	Enable/disable 60x bus address parity checking 0 Prevents address parity checking. 1 Allows a address parity error to cause a checkstop if MSR[ME] = 0 or a machine check exception if MSR[ME] = 1. EBA and EBD allow the processor to operate with memory subsystems that do not generate parity.
3	EBD	Enable 60x bus data parity checking 0 Parity checking is disabled. 1 Allows a data parity error to cause a checkstop if MSR[ME] = 0 or a machine check exception if MSR[ME] = 1. EBA and EBD allow the processor to operate with memory subsystems that do not generate parity.
4	BCLK	Reserved. Must set to 0.
5	—	Not used. Defined as EICE on some earlier processors.
6	ECLK	Reserved. Must set to 0.
7	PAR	Disable precharge of $\overline{\text{ARTRY}}$. 0 Precharge of $\overline{\text{ARTRY}}$ enabled 1 Alters bus protocol slightly by preventing the processor from driving $\overline{\text{ARTRY}}$ to high (negated) state. If this is done, the system must restore the signals to the high state.
8	DOZE	Doze mode enable. Operates in conjunction with MSR[POW]. 0 Doze mode disabled. 1 Doze mode enabled. Doze mode is invoked by setting MSR[POW] while this bit is set. In doze mode, the PLL, time base, and snooping remain active.

Table 2-4. HID0 Bit Functions (Continued)

Bit	Name	Function
9	NAP	Nap mode enable. Operates in conjunction with MSR[POW]. 0 Nap mode disabled. 1 Nap mode enabled. Doze mode is invoked by setting MSR[POW] while this bit is set. In nap mode, the PLL and the time base remain active.
10	SLEEP	Sleep mode enable. Operates in conjunction with MSR[POW]. 0 Sleep mode disabled. 1 Sleep mode enabled. Sleep mode is invoked by setting MSR[POW] while this bit is set. $\overline{\text{QREQ}}$ is asserted to indicate that the processor is ready to enter sleep mode. If the system logic determines that the processor may enter sleep mode, the quiesce acknowledge signal, $\overline{\text{QACK}}$, is asserted back to the processor. Once $\overline{\text{QACK}}$ assertion is detected, the processor enters sleep mode after several processor clocks. At this point, the system logic may turn off the PLL by first configuring PLL_CFG[0–3] to PLL bypass mode, then disabling SYSCLK.
11	DPM	Dynamic power management enable. 0 Dynamic power management is disabled. 1 Functional units may enter a low-power mode automatically if the unit is idle. This does not affect operational performance and is transparent to software or any external hardware.
12–14	—	Not used
15	NHR	Not hard reset (software-use only)—Helps software distinguish a hard reset from a soft reset. 0 A hard reset occurred if software had previously set this bit. 1 A hard reset has not occurred. If software sets this bit after a hard reset, when a reset occurs and this bit remains set, software can tell it was a soft reset.
16	ICE	Instruction cache enable 0 The instruction cache is neither accessed nor updated. All pages are accessed as if they were marked cache-inhibited (WIM = X1X). Potential cache accesses from the bus (snoop and cache operations) are ignored. In the disabled state for the L1 caches, the cache tag state bits are ignored and all accesses are propagated to the L2 cache or bus as single-beat transactions. For those transactions, however, $\overline{\text{CI}}$ reflects the original state determined by address translation regardless of cache disabled status. ICE is zero at power-up. 1 The instruction cache is enabled
17	DCE	Data cache enable 0 The data cache is neither accessed nor updated. All pages are accessed as if they were marked cache-inhibited (WIM = X1X). Potential cache accesses from the bus (snoop and cache operations) are ignored. In the disabled state for the L1 caches, the cache tag state bits are ignored and all accesses are propagated to the L2 cache or bus as single-beat transactions. For those transactions, however, $\overline{\text{CI}}$ reflects the original state determined by address translation regardless of cache disabled status. DCE is zero at power-up. 1 The data cache is enabled.
18	ILOCK	Instruction cache lock 0 Normal operation 1 Instruction cache is locked. A locked cache supplies data normally on a hit, but are treated as a cache-inhibited transaction on a miss. On a miss, the transaction to the bus or the L2 cache is single-beat, however, $\overline{\text{CI}}$ still reflects the original state as determined by address translation independent of cache locked or disabled status. To prevent locking during a cache access, an **isync** instruction must precede the setting of ILOCK.

Table 2-4. HID0 Bit Functions (Continued)

Bit	Name	Function
19	DLOCK	Data cache lock. 0 Normal operation 1 Data cache is locked. A locked cache supplies data normally on a hit but is treated as a cache-inhibited transaction on a miss. On a miss, the transaction to the bus or the L2 cache is single-beat, however, $\overline{CI}$ still reflects the original state as determined by address translation independent of cache locked or disabled status. A snoop hit to a locked L1 data cache performs as if the cache were not locked. A cache block invalidated by a snoop remains invalid until the cache is unlocked. To prevent locking during a cache access, a **sync** instruction must precede the setting of DLOCK.
20	ICFI	Instruction cache flash invalidate 0 The instruction cache is not invalidated. The bit is cleared when the invalidation operation begins (usually the next cycle after the write operation to the register). The instruction cache must be enabled for the invalidation to occur. 1 An invalidate operation is issued that marks the state of each instruction cache block as invalid without writing back modified cache blocks to memory. Cache access is blocked during this time. Bus accesses to the cache are signaled as a miss during invalidate-all operations. Setting ICFI clears all the valid bits of the blocks and the PLRU bits to point to way L0 of each set. Once the L1 flash invalidate bits are set through a **mtspr** operations, hardware automatically resets these bits in the next cycle (provided that the corresponding cache enable bits are set in HID0). Note, in the PowerPC 603 and PowerPC 603e processors, the proper use of the ICFI and DCFI bits was to set them and clear them in two consecutive **mtspr** operations. Software that already has this sequence of operations does not need to be changed to run on Gekko.
21	DCFI	Data cache flash invalidate 0 The data cache is not invalidated. The bit is cleared when the invalidation operation begins (usually the next cycle after the write operation to the register). The data cache must be enabled for the invalidation to occur. 1 An invalidate operation is issued that marks the state of each data cache block as invalid without writing back modified cache blocks to memory. Cache access is blocked during this time. Bus accesses to the cache are signaled as a miss during invalidate-all operations. Setting DCFI clears all the valid bits of the blocks and the PLRU bits to point to way L0 of each set. Once the L1 flash invalidate bits are set through a **mtspr** operations, hardware automatically resets these bits in the next cycle (provided that the corresponding cache enable bits are set in HID0). Setting this bit clears all the valid bits of the blocks and the PLRU bits to point to way L0 of each set. Note, In the PowerPC 603 and PowerPC 603e processors, the proper use of the ICFI and DCFI bits was to set them and clear them in two consecutive **mtspr** operations. Software that already has this sequence of operations does not need to be changed to run on Gekko.
22	SPD	Speculative cache access disable 0 Speculative bus accesses to nonguarded space (G = 0) from both the instruction and data caches is enabled 1 Speculative bus accesses to nonguarded space in both caches is disabled
23	IFEM	Enable M bit on bus for instruction fetches. 0 M bit disabled. Instruction fetches are treated as nonglobal on the bus 1 Instruction fetches reflect the M bit from the WIM settings.
24	SGE	Store gathering enable 0 Store gathering is disabled 1 Integer store gathering is performed for write-through to nonguarded space or for cache-inhibited stores to nonguarded space for 4-byte, word-aligned stores. The LSU combines stores to form a double word that is sent out on the 60x bus as a single-beat operation. Stores are gathered only if successive, eligible stores, are queued and pending. Store gathering is performed regardless of address order or endian mode.

Table 2-4. HID0 Bit Functions (Continued)

Bit	Name	Function
25	DCFA	Data cache flush assist. (Force data cache to ignore invalid sets on miss replacement selection.) 0 The data cache flush assist facility is disabled 1 The miss replacement algorithm ignores invalid entries and follows the replacement sequence defined by the PLRU bits. This reduces the series of uniquely addressed load or **dcbz** instructions to eight per set. The bit should be set just before beginning a cache flush routine and should be cleared when the series of instructions is complete.
26	BTIC	Branch Target Instruction Cache enable—used to enable use of the 64-entry branch instruction cache. 0 The BTIC is disabled, the contents are invalidated, and the BTIC behaves as if it was empty. New entries cannot be added until the BTIC is enabled. 1 The BTIC is enabled, and new entries can be added.
27	—	Not used. Defined as FBIOB on earlier 603-type processors.
28	ABE	Address broadcast enable—controls whether certain address-only operations (such as cache operations, **eieio**, and **sync**) are broadcast on the 60x bus. 0 Address-only operations affect only local L1 and L2 caches and are not broadcast. 1 Address-only operations are broadcast on the 60x bus.Affected instructions are **eieio**, **sync**, **dcbi**, **dcbf**, and **dcbst**. A **sync** instruction completes only after a successful broadcast. Execution of **eieio** causes a broadcast that may be used to prevent any external devices, such as a bus bridge chip, from store gathering. Note that **dcbz** (with M = 1, coherency required) always broadcasts on the 60x bus regardless of the setting of this bit. An **icbi** is never broadcast. No cache operations, except **dcbz**, are snooped by Gekko regardless of whether the ABE is set. Bus activity caused by these instructions results directly from performing the operation on the Gekko cache.
29	BHT	Branch history table enable 0 BHT disabled. Gekko uses static branch prediction as defined by the PowerPC architecture (UISA) for those branch instructions the BHT would have otherwise used to predict (that is, those that use the CR as the only mechanism to determine direction). For more information on static branch prediction, see “Conditional Branch Control,” in Chapter 4 of the *PowerPC Microprocessor Family: The Programming Environments* manual. 1 Allows the use of the 512-entry branch history table (BHT). The BHT is disabled at power-on reset. All entries are set to weakly, not-taken.
30	—	Not used
31	NOOPTI	No-op the data cache touch instructions. 0 The **dcbt** and **dcbtst** instructions are enabled. 1 The **dcbt** and **dcbtst** instructions are no-oped globally.

HID0 can be accessed with **mtspr** and **mfspr** using SPR1008.

2.1.2.3 Hardware Implementation-Dependent Register 1

The hardware implementation-dependent register 1 (HID1) reflects the state of the PLL_CFG[0–3] signals. The HID1 bits are shown in Figure 2-4.

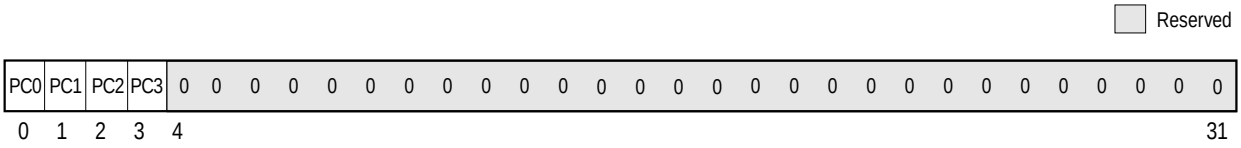


Figure 2-4. Hardware Implementation-Dependent Register 1 (HID1)

The HID1 bits are described in Table 2-5.

Table 2-5. HID1 Bit Functions

Bit(s)	Name	Description
0	PC0	PLL configuration bit 0 (read-only)
1	PC1	PLL configuration bit 1 (read-only)
2	PC2	PLL configuration bit 2 (read-only)
3	PC3	PLL configuration bit 3 (read-only)
4–31	—	Reserved

Note: The clock configuration bits reflect the state of the PLL_CFG[0–3] signals.

HID1 can be accessed with **mtspr** and **mfspr** using SPR 1009.

2.1.2.4 Hardware Implementation-Dependent Register 2

The hardware implementation-dependent register 2 (HID2) controls the state of the graphics enhancement features in Gekko. The HID2 register is shown in Figure 2-5.

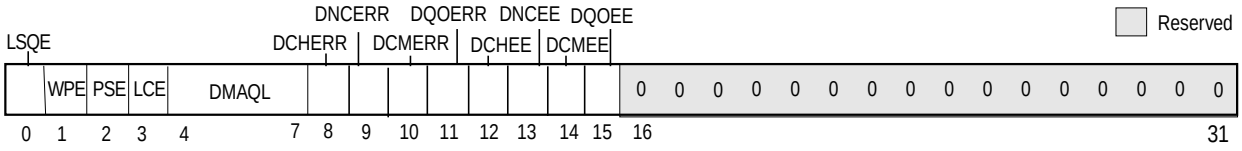


Figure 2-5. Hardware Implementation-Dependent Register 2 (HID2)

The HID2 bits are described in Table 2-6

Table 2-6. HID2 Bit Settings

Bit	Name	Function
0	LSQE	Load/Store quantized enable for non-indexed format instructions (**psq_l**, **psq_lu**, **psq_st**, **psq_stu**).
1	WPE	Write pipe enable. 0 Write gathering is disabled. 1 Write gather pipe is enabled. Non-cacheable stores to the WPAR address are gathered and transferred in 32 byte blocks over the 60x bus.
2	PSE	Paired single enable. 0 All paired single instructions are illegal. 1 Paired single instructions can be used.
3	LCE	Locked cache enable. 0 Cache is not partitioned. Data cache is 32 Kbytes. **dcbz_l** instruction is illegal. DMA facility is disabled. 1 Data cache is partitioned into 16 Kbytes of normal cache and 16 Kbytes of locked cache. **dcbz_l** instruction will allocate lines in the locked cache. DMA facility can be used to move data between the locked cache and external memory. In Gekko, locked cache and bus snoop are incompatible. LCE shall be kept at 0 for systems which generate snoop transactions.

Table 2-6. HID2 Bit Settings

4-7	DMAQL	DMA queue length (read only). The DMAQL value indicates the number of DMA commands outstanding. A value of zero indicates an empty DMA command queue. A value of 15 indicates the DMA command queue is full.
8	DCHERR	**dcbz_l** cache hit error (sticky).
9	DNCERR	DMA access to normal cache error (sticky).
10	DCMERR	DMA cache miss error (sticky).
11	DQOERR	DMA queue overflow error (sticky).
12	DCHEE	**dcbz_l** cache hit error enable.
13	DNCEE	DMA access to normal cache error enable.
14	DCMEE	DMA cache miss error enable.
15	DQOEE	DMA queue overflow error enable.
16-31	—	Reserved.

HID2 can be accessed with **mtspr** and **mfspr** using SPR 920.

When using **mtspr** to set any of the three enable bits, LSQE, PSE and LCE, the i-cache must be invalidated before using any of the corresponding Gekko graphics extension instructions.

2.1.2.5 Performance Monitor Registers

This section describes the registers used by the performance monitor, which is described in Chapter 11, "Performance Monitor" in this manual.

2.1.2.5.1 Monitor Mode Control Register 0 (MMCR0)

The monitor mode control register 0 (MMCR0), shown in Figure 2-6, is a 32-bit SPR provided to specify events to be counted and recorded. The MMCR0 can be accessed only in supervisor mode. User-level software can read the contents of MMCR0 by issuing an **mfspr** instruction to UMMCR0, described in the next section.

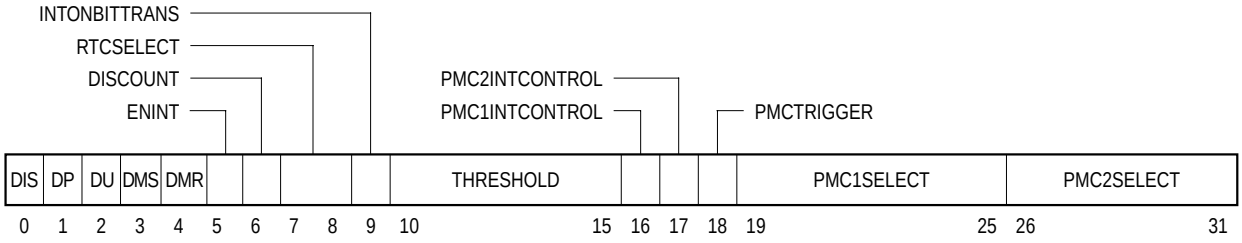


Figure 2-6. Monitor Mode Control Register 0 (MMCR0)

This register must be cleared at power up. Reading this register does not change its contents. The bits

of the MMCR0 register are described in Table 2-7.

Table 2-7. MMCR0 Bit Settings

Bit	Name	Description
0	DIS	Disables counting unconditionally 0 The values of the PMC*n* counters can be changed by hardware. 1 The values of the PMC*n* counters cannot be changed by hardware.
1	DP	Disables counting while in supervisor mode 0 The PMC*n* counters can be changed by hardware. 1 If the processor is in supervisor mode (MSR[PR] is cleared), the counters are not changed by hardware.
2	DU	Disables counting while in user mode 0 The PMC*n* counters can be changed by hardware. 1 If the processor is in user mode (MSR[PR] is set), the PMC*n* counters are not changed by hardware.
3	DMS	Disables counting while MSR[PM] is set 0 The PMC*n* counters can be changed by hardware. 1 If MSR[PM] is set, the PMC*n* counters are not changed by hardware.
4	DMR	Disables counting while MSR(PM) is zero. 0 The PMC*n* counters can be changed by hardware. 1 If MSR[PM] is cleared, the PMC*n* counters are not changed by hardware.
5	ENINT	Enables performance monitor interrupt signaling. 0 Interrupt signaling is disabled. 1 Interrupt signaling is enabled. Cleared by hardware when a performance monitor interrupt is signaled. To reenable these interrupt signals, software must set this bit after handling the performance monitor interrupt. The IPL ROM code clears this bit before passing control to the operating system.
6	DISCOUNT	Disables counting of PMC*n* when a performance monitor interrupt is signaled (that is, ((PMC*n*INTCONTROL = 1) & (PMC*n*[0] = 1) & (ENINT = 1)) or the occurrence of an enabled time base transition with ((INTONBITTRANS =1) & (ENINT = 1)). 0 Signaling a performance monitor interrupt does not affect counting status of PMC*n*. 1 The signaling of a performance monitor interrupt prevents changing of PMC1 counter. The PMC*n* counter do not change if PMC2COUNTCTL = 0. Because a time base signal could have occurred along with an enabled counter overflow condition, software should always reset INTONBITTRANS to zero, if the value in INTONBITTRANS was a one.
7–8	RTCSELECT	64-bit time base, bit selection enable 00 Pick bit 63 to count 01 Pick bit 55 to count 10 Pick bit 51 to count 11 Pick bit 47 to count
9	INTONBITTRANS	Cause interrupt signaling on bit transition (identified in RTCSELECT) from off to on 0 Do not allow interrupt signal if chosen bit transitions. 1 Signal interrupt if chosen bit transitions. Software is responsible for setting and clearing INTONBITTRANS.
10–15	THRESHOLD	Threshold value. Gekko supports all 6 bits, allowing threshold values from 0–63. The intent of the THRESHOLD support is to characterize L1 data cache misses.

Table 2-7. MMCR0 Bit Settings (Continued)

Bit	Name	Description
16	PMC1INTCONTROL	Enables interrupt signaling due to PMC1 counter overflow. 0 Disable PMC1 interrupt signaling due to PMC1 counter overflow 1 Enable PMC1 Interrupt signaling due to PMC1 counter overflow
17	PMCINTCONTROL	Enable interrupt signaling due to any PMC2–PMC4 counter overflow. Overrides the setting of DISCOUNT. 0 Disable PMC2–PMC4 interrupt signaling due to PMC2–PMC4 counter overflow. 1 Enable PMC2–PMC4 interrupt signaling due to PMC2–PMC4 counter overflow.
18	PMCTRIGGER	Can be used to trigger counting of PMC2–PMC4 after PMC1 has overflowed or after a performance monitor interrupt is signaled. 0 Enable PMC2–PMC4 counting. 1 Disable PMC2–PMC4 counting until either PMC1[0] = 1 or a performance monitor interrupt is signaled.
19–25	PMC1SELECT	PMC1 input selector, 128 events selectable. See Table 2-9.
26–31	PMC2SELECT	PMC2 input selector, 64 events selectable. See Table 2-9.

MMCR0 can be accessed with **mtspr** and **mfspr** using SPR 952.

2.1.2.5.2 User Monitor Mode Control Register 0 (UMMCR0)

The contents of MMCR0 are reflected to UMMCR0, which can be read by user-level software. MMCR0 can be accessed with **mfspr** using SPR 936.

2.1.2.5.3 Monitor Mode Control Register 1 (MMCR1)

The monitor mode control register 1 (MMCR1) functions as an event selector for performance monitor counter registers 3 and 4 (PMC3 and PMC4). The MMCR1 register is shown in Figure 2-7.

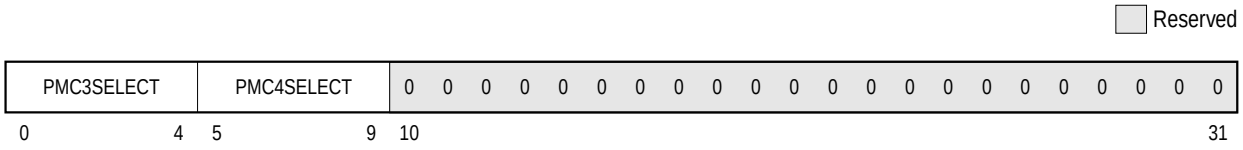


Figure 2-7. Monitor Mode Control Register 1 (MMCR1)

Bits for MMCR1 are shown in Table 2-8; the corresponding events are described in Section 2.1.2.5.5 below.

Table 2-8. MMCR1 Bits

Bits	Name	Description
0–4	PMC3SELECT	PMC3 input selector. 32 events selectable. See Table 2-9for defined selections.
5–9	PMC4SELECT	PMC4 input selector. 32 events selectable. See Table 2-9for defined selections.
10–31	—	Reserved

MMCR1 can be accessed with **mtspr** and **mfspr** using SPR 956. User-level software can read the contents of MMCR1 by issuing an **mfspr** instruction to UMMCR1, described next.

2.1.2.5.4 User Monitor Mode Control Register 1 (UMMCR1)

The contents of MMCR1 are reflected to UMMCR1, which can be read by user-level software. MMCR1 can be accessed with **mfspr** using SPR 940.

2.1.2.5.5 Performance Monitor Counter Registers (PMC1–PMC4)

PMC1–PMC4, shown in Figure 2-8, are 32-bit counters that can be programmed to generate interrupt signals when they overflow.

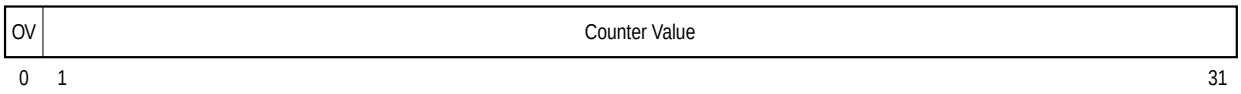


Figure 2-8. Performance Monitor Counter Registers (PMC1–PMC4)

The bits contained in the PMC*n* registers are described in Table 2-9.

Table 2-9. PMC*n* Bits

Bits	Name	Description
0	OV	Overflow. When this bit is set it indicates that this counter has reached its maximum value.
1–31	Counter value	Indicates the number of occurrences of the specified event.

Counters are considered to overflow when the high-order bit (the sign bit) becomes set; that is, they reach the value 2147483648 (0x8000_0000). However, an interrupt is not signaled unless both PMC*n*[INTCONTROL] and MMCR0[ENINT] are also set.

Note that the interrupts can be masked by clearing MSR[EE]; the interrupt signal condition may occur with MSR[EE] cleared, but the exception is not taken until EE is set. Setting MMCR0[DISCOUNT] forces counters to stop counting when a counter interrupt occurs.

Software is expected to use **mtspr** to set PMC explicitly to nonoverflow values. If software sets an overflow value, an erroneous exception may occur. For example, if both PMC*n*[INTCONTROL] and MMCR0[ENINT] are set and **mtspr** loads an overflow value, an interrupt signal may be generated without any event counting having taken place.

The event to be monitored by PMC1 can be chosen by setting MMCR0[19–25]. The event to be monitored by PMC2 can be chosen by setting MMCR0[26-31]. The event to be monitored by PMC3 can be chosen by setting MMCR1[0-4]. The event to be monitored by PMC4 can be chosen by setting MMCR1[5-9]. The selected events are counted beginning when MMCR0 is set until either MMCR0 is reset or a performance monitor interrupt is generated.

Table 11-5 on Page 11-6, Table 11-6 on Page 11-7, Table 11-7 on Page 11-8, and Table 11-8 on Page 11-9 list the selectable events and their encodings.

The PMC registers can be accessed with **mtspr** and **mfspr** using following SPR numbers:

- PMC1 is SPR 953
- PMC2 is SPR 954
- PMC3 is SPR 957
- PMC4 is SPR 958

2.1.2.5.6 User Performance Monitor Counter Registers (UPMC1–UPMC4)

The contents of the PMC1–PMC4 are reflected to UPMC1–UPMC4, which can be read by user-level software. The UPMC registers can be read with **mfspr** using the following SPR numbers:

- UPMC1 is SPR 937
- UPMC2 is SPR 938
- UPMC3 is SPR 941
- UPMC4 is SPR 942

2.1.2.5.7 Sampled Instruction Address Register (SIA)

The sampled instruction address register (SIA) is a supervisor-level register that contains the effective address of an instruction executing at or around the time that the processor signals the performance monitor interrupt condition. The SIA is shown in Figure 2-9.

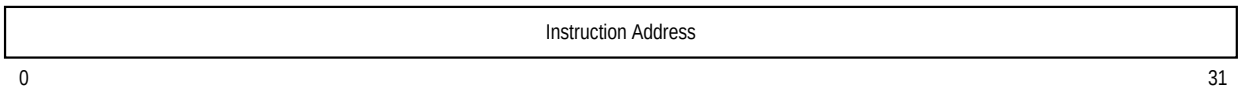


Figure 2-9. Sampled Instruction Address Registers (SIA)

If the performance monitor interrupt is triggered by a threshold event, the SIA contains the exact instruction (called the sampled instruction) that caused the counter to overflow.

If the performance monitor interrupt was caused by something besides a threshold event, the SIA contains the address of the last instruction completed during that cycle. SIA can be accessed with the **mtspr** and **mfspr** instructions using SPR 955.

2.1.2.5.8 User Sampled Instruction Address Register (USIA)

The contents of SIA are reflected to USIA, which can be read by user-level software. USIA can be accessed with the **mfspr** instructions using SPR 939.

2.1.2.5.9 Sampled Data Address Register (SDA) and User Sampled Data Address Register (USDA)

Gekko does not implement the sampled data address register (SDA) or the user-level, read-only USDA registers. However, for compatibility with processors that do, those registers can be written to

by boot code without causing an exception. SDA is SPR 959; USDA is SPR 943.

2.1.2.6 Instruction Cache Throttling Control Register (ICTC)

Reducing the rate of instruction fetching can control junction temperature without the complexity and overhead of dynamic clock control. System software can control instruction forwarding by writing a nonzero value to the ICTC register, a supervisor-level register shown in Figure 2-10. The overall junction temperature reduction comes from the dynamic power management of each functional unit when Gekko is idle in between instruction fetches. PLL (phase-locked loop) and DLL (delay-locked loop) configurations are unchanged.

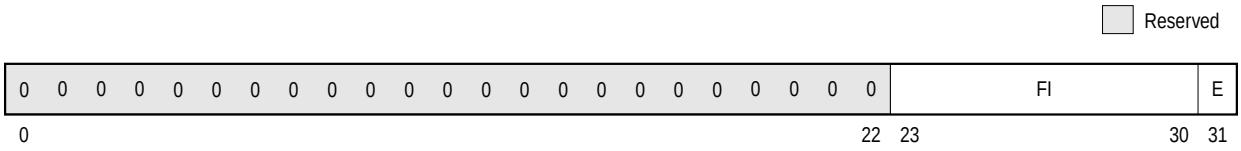


Figure 2-10. Instruction Cache Throttling Control Register (ICTC)

Table 2-10 describes the bit fields for the ICTC register.

Table 2-10. ICTC Bit Settings

Bits	Name	Description
0–22	—	Reserved
23–30	FI	Instruction forwarding interval expressed in processor clocks. 0x00 0 clock cycle. 0x01 1 clock cycle : 0xFF 255 clock cycles
31	E	Cache throttling enable 0 Disable instruction cache throttling. 1 Enable instruction cache throttling.

Instruction cache throttling is enabled by setting ICTC[E] and writing the instruction forwarding interval into ICTC[FI]. Enabling, disabling, and changing the instruction forwarding interval affect instruction forwarding immediately.

The ICTC register can be accessed with the **mtspr** and **mfspr** instructions using SPR 1019.

2.1.2.7 Thermal Management Registers (THRM1–THRM3)

The on-chip thermal management assist unit provides the following functions:

- Compares the junction temperature against user programmed thresholds
- Generates a thermal management interrupt if the temperature crosses the threshold
- Provides a way for a successive approximation routine to estimate junction temperature

Control and access to the thermal management assist unit is through the privileged **mtspr**/**mfspr** instructions to the three THRM registers. THRM1 and THRM2, shown in Figure 2-11, provide the ability to compare the junction temperature against two user-provided thresholds. Having dual thresholds allows thermal management software differing degrees of action in reducing junction temperature. Thermal management can use a single-threshold mode in which the thermal sensor

output is compared to only one threshold in either THRM1 or THRM2.

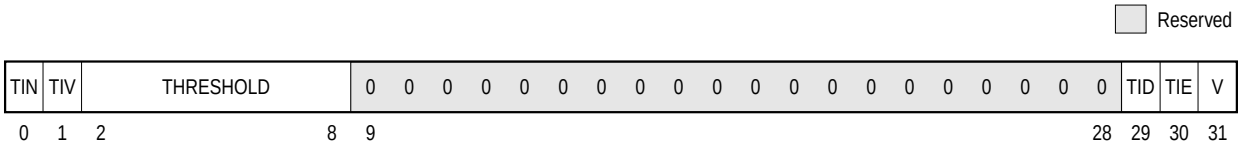


Figure 2-11. Thermal Management Registers 1–2 (THRM1–THRM2)

The bits in THRM1 and THRM2 are described in Table 2-11.

Table 2-11. THRM1–THRM2 Bit Settings

Bits	Field	Description
0	TIN	Thermal management interrupt bit. Read-only. This bit is set if the thermal sensor output crosses the threshold specified in the SPR. The state of TIN is valid only if TIV is set. The interpretation of TIN is controlled by TID. See Table 2-12.
1	TIV	Thermal management interrupt valid. Read-only. This bit is set by the thermal assist logic to indicate that the thermal management interrupt (TIN) state is valid. See Table 2-12.
2–8	Threshold	Threshold that the thermal sensor output is compared to. The range is 0 —127 Çand each bit represents 1 C Note that this is not the resolution of the thermal sensor.
9–28	—	Reserved. System software should clear these bits when writing to the THRM*n* SPRs.
29	TID	Thermal management interrupt direction bit. Selects the result of the temperature comparison to set TIN and to assert a thermal management interrupt if TIE is set. If TID is cleared, TIN is set and an interrupt occurs if the junction temperature exceeds the threshold. If TID is set, TIN is set and an interrupt is indicated if the junction temperature is below the threshold. See Table 2-16 on Page 2-24.
30	TIE	Thermal management interrupt enable. The thermal management interrupt is maskable by the MSR[EE] bit. If TIE is cleared and THRM*n* is valid, the TIN bit records the status of the junction temperature vs. threshold comparison without causing an exception. This lets system software successively approximate the junction temperature. See Table 2-16 on Page 2-24.
31	V	SPR valid bit. Setting this bit indicates the SPR contains a valid threshold, TID and TIE controls bits. THRM1/2[V] = 1 and THRM3[E] = 1 enables the thermal sensor operation. See Table 2-16 on Page 2-24.

If an **mtspr** affects a THRM register that contains operating parameters for an ongoing comparison during operation of the thermal assist unit, the respective TIV bits are cleared and the comparison is restarted. Changing THRM3 forces the TIV bits of both THRM1 and THRM2 to 0, and restarts the comparison if THRM3[E] is set.

Examples of valid THRM1/THRM2 bit settings are shown in Table 2-12.

Table 2-12. Valid THRM1/THRM2 Bit Settings

TIN[1]	TIV[1]	TID	TIE	V	Description
x	x	x	x	0	Invalid entry. The threshold in the SPR is not used for comparison.
x	x	x	0	1	Disable thermal management interrupt assertion.
x	x	0	x	1	Set TIN and assert thermal management interrupt if TIE = 1 and the junction temperature exceeds the threshold.
x	x	1	x	1	Set TIN and assert thermal management interrupt if TIE = 1 and the junction temperature is less than the threshold.
x	0	x	x	1	The state of the TIN bit is not valid.
0	1	0	x	1	The junction temperature is less than the threshold and as a result the thermal management interrupt is not generated for TIE = 1.
1	1	0	x	1	The junction temperature is greater than the threshold and as a result the thermal management interrupt is generated if TIE = 1.
0	1	1	x	1	The junction temperature is greater than the threshold and as a result the thermal management interrupt is not generated for TIE = 1.
1	1	1	x	1	The junction temperature is less than the threshold and as a result the thermal management interrupt is generated if TIE = 1.

Note:
[1] TIN and TIV are read-only status bits.

The THRM3 register, shown in Figure 2-12, is used to enable the thermal assist unit and to control the comparator output sample time. The thermal assist logic manages the thermal management interrupt generation and time-multiplexed comparisons in dual-threshold mode as well as other control functions.

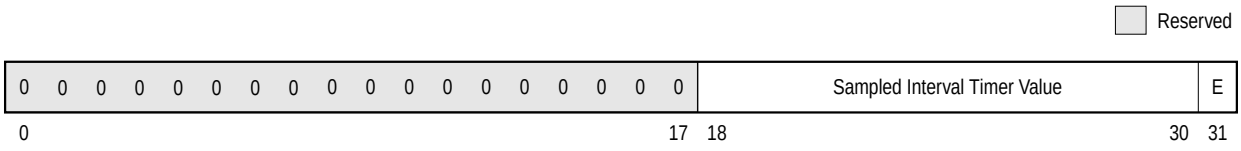


Figure 2-12. Thermal Management Register 3 (THRM3)

The bits in THRM3 are described in Table 2-13.

Table 2-13. THRM3 Bit Settings

Bits	Name	Description
0–17	—	Reserved for future use. System software should clear these bits when writing to the THRM3.
18–30	SITV	Sample interval timer value. Number of elapsed processor clock cycles before a junction temperature vs. threshold comparison result is sampled for TIN bit setting and interrupt generation. This is necessary due to the thermal sensor, DAC, and the analog comparator settling time being greater than the processor cycle time. The value should be configured to allow a sampling interval of 20 microseconds.
31	E	Enables the thermal sensor compare operation if either THRM1[V] or THRM2[V] is set.

The THRM registers can be accessed with the **mtspr** and **mfspr** instructions using the following SPR numbers:

- THRM1 is SPR 1020
- THRM2 is SPR 1021
- THRM3 is SPR 1022

2.1.2.8 Direct Memory Access (DMA) registers

The pair of DMA registers, DMAU and DMAL, is used to specify and issue a DMA command. A DMA command specifies the transfer of a contiguous block of data, up to 4 Kbytes, between the locked cache and external memory. Each DMA command consists of the starting address in locked cache, the starting address in external memory, the length of the transfer in cache lines, and the direction of the transfer.

The DMA facility is enabled using the HID2[LCE] bit. When HID2[LCE] = 0, the **mtspr** and **mfspr** instructions can be used to read and write the DMA registers, but the DMA commands associated with these registers will be ignored. In particular, the DMA_T and DMA_F bits in DMAL are always forced to zero in this mode. When HID2[LCE] = 1, a **mtspr** to DMAL with the DMA_F bit = 1 will cause the DMA command queue to be flushed, otherwise a **mtspr** DMAL with the DMA_T bit = 1 will cause the DMA command specified in the DMA registers to be added to the DMA command queue.

Figure 2-13 and Figure 2-14 on Page 2-23 show the format of the upper and lower DMA registers.

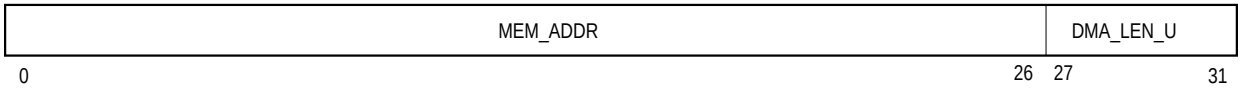


Figure 2-13. Direct Memory Access Upper (DMAU) register

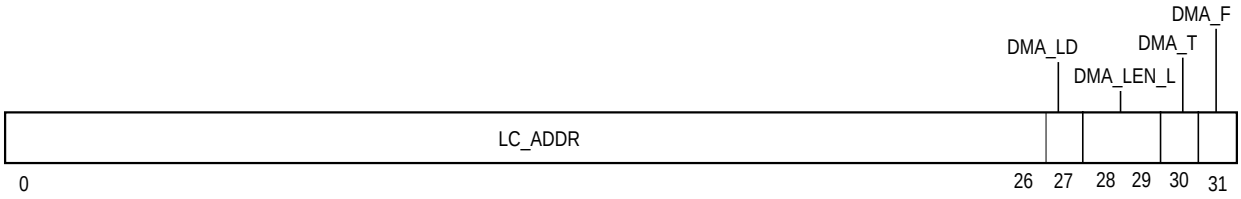


Figure 2-14. Direct Memory Access Lower (DMAL) register

Table 2-14 and Table 2-15 describe the bit fields for the DMA registers.

Table 2-14. DMAU Bit Settings

Bits	Name	Description
0–26	MEM_ADDR	High order address bits of starting address in external memory of the DMA transfer. The low order address bits are zero, forcing the starting address to be cache line aligned.
27–31	DMA_LEN_U	High order bits of transfer length, in cache lines. Low order bits are in DMAL.

Table 2-15. DMAL Bit Settings

Bits	Name	Description
0–26	LC_ADDR	High order address bits of starting address in locked cache of the DMA transfer. The low order address bits are zero, forcing the starting address to be cache line aligned.
27	DMA_LD	DMA load command 0 Store - transfer from locked cache to external memory 1 Load - transfer from external memory to locked cache
28–29	DMA_LEN_L	Low order bits of transfer length, in cache lines. High order bits are in DMAU.
30	DMA_T	Trigger bit 0 DMA command inactive. 1 **mtspr** DMAL instruction with this bit active will enqueue this DMA command.
31	DMA_F	Flush bit 0 Normal DMA operation. 1 **mtspr** DMAL instruction with this bit active will flush the DMA queue.

DMAU can be accessed with **mtspr** and **mfspr** using SPR 922. DMAL can be accessed with **mtspr** and **mfspr** using SPR 923.

2.1.2.9 Graphics Quantization Registers (GQRs)

The eight graphics quantization registers, GQR0 to GQR7, are used to specify the data type and scaling factor used to convert operands in paired single quantized load and store instructions. The specific GQR used for a particular instruction is specified by the three bit I field in the instruction. Figure 2-15 shows the format of a GQR.

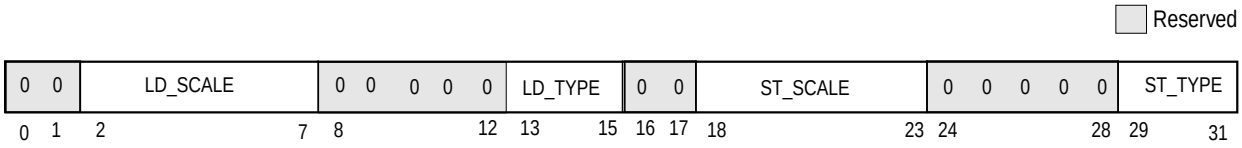


Figure 2-15. Graphics Quantization Register

Table 2-16 describes the bit fields for the GQR registers, and Table 2-17 lists the encoding of the type fields in the GQR for the various quantized data types.

Table 2-16. Graphics Quantization Register Bit Settings

Bits	Name	Description
0–1	—	Reserved
2-7	LD_SCALE	Scale value used by a load instruction.
8-12	—	Reserved
13-15	LD_TYPE	Type of operand in memory to be converted by a load instruction. See Table 2-22 on Page 2-31.
16-17	—	Reserved
18-23	ST_SCALE	Scale value used by a store instruction.
24-28	—	Reserved
29-31	ST_TYPE	Type of operand resulting from a conversion by a store instruction. See Table 2-22 on Page 2-31.

Table 2-17. Quantized Data Types

Code	Type
0	single-precision floating-point (no conversion)
1-3	reserved
4	unsigned 8 bit integer
5	unsigned 16 bit integer
6	signed 8 bit integer
7	signed 16 bit integer

GQR0 through GQR7 can be accessed with **mtspr** and **mfspr** using SPR 912 through 919, respectively.

2.1.2.10 Write Pipe Address Register (WPAR)

The write pipe address register, shown in Figure 2-16 holds the physical address of operands to be gathered by the write gather pipe facility. A **mtspr** to the WPAR establishes the gather address and

resets the state of the facility, discarding any data in the buffer. A **mfspr** WPAR is used to read the BNE bit to check for any outstanding data transfers.

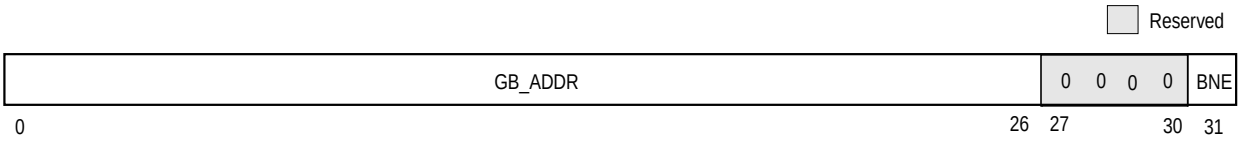


Figure 2-16. Write Pipe Address Register (WPAR)

Table 2-18 describes the bit fields for the WPAR register.

Table 2-18. Write Pipe Address Register Bit Settings

Bits	Name	Description
0–26	GB_ADDR	High order address bits of the data to be gathered. The low order address bits are zero, forcing the address to be cache line aligned. Note that only these 27 bits are compared to determine if a non-cacheable store will be gathered. If the address of the non-cacheable store has a non-zero value in the low order five bits, incorrect data will be gathered.
27–30	—	Reserved
31	BNE	Buffer not empty (read only)

WPAR can be accessed with **mtspr** and **mfspr** using SPR 921.

2.1.2.11 L2 Cache Control Register (L2CR)

The L2 cache control register, shown in Figure 2-17, is a supervisor-level, implementation-specific SPR used to configure and operate the L2 cache. It is cleared by a hard reset or power-on reset.

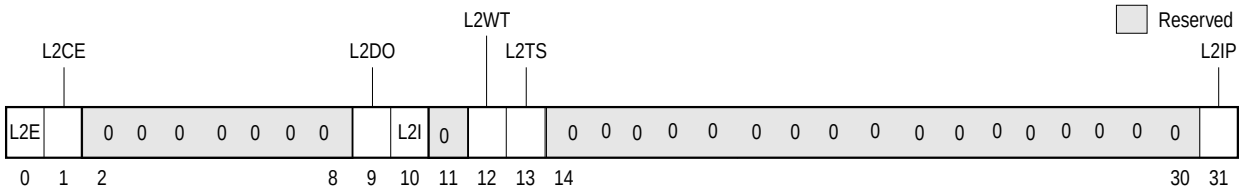


Figure 2-17. L2 Cache Control Register (L2CR)

The L2 cache interface is described in Chapter 9, "L2 Cache, Locked D-Cache, DMA and Write Gather Pipe" in this manual,

The L2CR bits are described in Table 2-19.

Table 2-19. L2CR Bit Settings

Bit	Name	Function
0	L2E	L2 enable. Enables L2 cache operation (including snooping) starting with the next transaction the L2 cache unit receives. Before enabling the L2 cache, all other L2CR bits must be set appropriately. The L2 cache may need to be invalidated globally.

Table 2-19. L2CR Bit Settings (Continued)

Bit	Name	Function
1	L2CE	L2 Checkstop enable 0 ECC double bit error does not cause a Machine Check. 1 ECC double bit error causes a machine check exception.
2–8	—	Reserved
9	L2DO	L2 data-only. Setting this bit enables data-only operation in the L2 cache. For this operation, only transactions from the L1 data cache can be cached in the L2 cache, which treats all transactions from the L1 instruction cache as cache-inhibited (bypass L2 cache, no L2 checking done). This bit is provided for L2 testing only.
10	L2I	L2 global invalidate. Setting L2I invalidates the L2 cache globally by clearing the L2 bits including status bits. This bit must not be set while the L2 cache is enabled.
11	—	Reserved
12	L2WT	L2 write-through. Setting L2WT selects write-through mode (rather than the default write-back mode) so all writes to the L2 cache also write through to the 60x bus. For these writes, the L2 cache entry is always marked as clean (valid unmodified) rather than dirty (valid modified). This bit must never be asserted after the L2 cache has been enabled as previously-modified lines can get remarked as clean during normal operation.
13	L2TS	L2 test support. Setting L2TS causes cache block pushes from the L1 data cache that result from **dcbf** and **dcbst** instructions to be written only into the L2 cache and marked valid, rather than being written only to the 60x bus and marked invalid in the L2 cache in case of hit. This bit allows a **dcbz**/**dcbf** instruction sequence to be used with the L1 cache enabled to easily initialize the L2 cache with any address and data information. This bit also keeps **dcbz** instructions from being broadcast on the 60x and single-beat cacheable store misses in the L2 from being written to the 60x bus.
14–30	—	Reserved
31	L2IP	L2 global invalidate in progress (read only). This read-only bit indicates whether an L2 global invalidate is occurring. It should be monitored after an L2 global invalidate has been initiated by the L2I bit to determine when it has completed.

The L2CR register can be accessed with the **mtspr** and **mfspr** instructions using SPR 1017.

2.2 Operand Conventions

This section describes the operand conventions as they are represented in two levels of the PowerPC architecture—UISA and VEA. Detailed descriptions of conventions used for storing values in registers and memory, accessing PowerPC registers, and representation of data in these registers can be found in Chapter 3, “Operand Conventions" in the *PowerPC Microprocessor Family: The Programming Environments* manual.

2.2.1 Data Organization in Memory and Data Transfers

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

Memory operands may be bytes, half words, words, or double words, or, for the load/store multiple and load/store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction.

2.2.2 Alignment and Misaligned Accesses

The operand of a single-register memory access instruction has an alignment boundary equal to its length. An operand’s address is misaligned if it is not a multiple of its width. Operands for single-register memory access instructions have the characteristics shown in Table 2-20. Although not permitted as memory operands, quad words are shown because quad-word alignment is desirable for certain memory operands.

Table 2-20. Memory Operands

Operand	Length	Addr[28-31] If Aligned
Byte	8 bits	xxxx
Half word	2 bytes	xxx0
Word	4 bytes	xx00
Double word	8 bytes	x000
Quad word	16 bytes	0000

Note: An “x” in an address bit position indicates that the bit can be 0 or 1 independent of the state of other bits in the address.

The concept of alignment is also applied more generally to data in memory. For example, a 12-byte data item is said to be word-aligned if its address is a multiple of four.

Some instructions require their memory operands to have certain alignment. In addition, alignment may affect performance. For single-register memory access instructions, the best performance is obtained when memory operands are aligned.

Instructions are 32 bits (one word) long and must be word-aligned.

Gekko does not provide hardware support for floating-point memory that is not word-aligned. If a floating-point operand is not aligned, Gekko invokes an alignment exception, and it is left up to software to break up the offending storage access operation appropriately. In addition, some non-double-word–aligned memory accesses suffer performance degradation as compared to an aligned access of the same type.

In general, floating-point word accesses should always be word-aligned and floating-point double-word accesses should always be double-word–aligned. Frequent use of misaligned

accesses is discouraged since they can degrade overall performance.

2.2.3 Floating-Point Operand and Execution Models—UISA

The IEEE 754 standard defines conventions for 64- and 32-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision arithmetic instructions to have either (or both) single-precision or double-precision operands, but states that single-precision arithmetic instructions should not accept double-precision operands.

The PowerPC UISA follows these guidelines:

- Double-precision arithmetic instructions may have single-precision operands but always produce double-precision results.
- Single-precision arithmetic instructions require all operands to be single-precision and always produce single-precision results.

For arithmetic instructions, conversion from double- to single-precision must be done explicitly by software, while conversion from single- to double-precision is done implicitly by the processor.

All PowerPC implementations provide the equivalent of the execution models described in Section 3.3 of the *PowerPC Microprocessor Family: The Programming Environments* manual to ensure that identical results are obtained. The definition of the arithmetic instructions for infinities, denormalized numbers, and NaNs follow conventions described in that section.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic uses two additional bit positions to avoid potential transient overflow conditions. An extra bit is required when denormalized double-precision numbers are prenormalized. A second bit is required to permit computation of the adjusted exponent value in the following examples when the corresponding exception enable bit is one:

- Underflow during multiplication using a denormalized operand
- Overflow during division using a denormalized divisor

Gekko provides hardware support for all single- and double-precision floating-point operations for most value representations and all rounding modes. This architecture provides for hardware to implement a floating-point system as defined in ANSI/IEEE standard 754-1985, *IEEE Standard for Binary Floating Point Arithmetic*. Detailed information about the floating-point execution model for non-paired single mode (HID2[PSE] = 0) can be found in Chapter 3, “Operand Conventions" in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Gekko supports non-IEEE mode whenever FPSCR[29] is set. In this mode, denormalized numbers, NaNs, and some IEEE invalid operations are treated in a non-IEEE conforming manner. This is accomplished by delivering results that approximate the values required by the IEEE standard.

In addition to single- and double-precision operands, Gekko supports a third format, called paired single, when HID2[PSE] = 1. (Note that HID2[PSE] can be changed only when the i-cache is invalidated and disabled.) Paired single operands are represented in the 64 bit floating-point registers as two 32 bit single-precision floating-point values.

We will refer to the single-precision floating-point value in the high order word as ps0, and that in the low order word as ps1.

Figure 2-18 shows the format of an FPR containing a paired single operand.

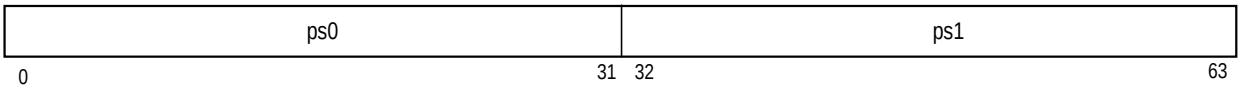


Figure 2-18. Floating-Point Register containing a paired single operand

Most of the new instructions for manipulating these operands allow both values to be processed in parallel in the execution unit. For example, the paired single multiply-add instruction (**ps_madd**) multiplies ps0 in frA by ps0 in frC, then adds it to ps0 in frB to get a result that is placed in ps0 in frD. Simultaneously, the same operations are applied to the corresponding ps1 values. Note that paired single instructions, including loads, stores and moves, cause a floating-point unavailable exception if execution is attempted when MSR[FP] = 0.

Many of the new paired single instructions perform an operation comparable to one of the existing double-precision instructions. For example, **fadd** adds double-precision operands from two registers and places the result into a third register. In the corresponding paired single instruction, **ps_add**, two such operations are performed in parallel, one on the ps0 values, and one on the ps1 values. Several other paired single instructions are supported that do not have exact analogs to existing double-precision instructions. See Chapter 12, "Instruction Set" in this manual for a detailed description of the paired single instructions.

Most paired single instructions produce a pair of result values. The Floating-Point Status and Control Register (FPSCR) contains a number of status bits that are affected by the floating-point computation. FPSCR bits 15-19 are the result bits. They are determined by the result of the ps0 computation, except for **ps_cmpu1**, **ps_cmpo1** and **ps_sum1** where the result bits are determined by the result of the ps1 computation.The FPSCR bits that reflect exceptional conditions in the computation are bits 0-14, and 22-23. For paired single instructions that affect any of these bits, either the ps0 or the ps1 computation can set the bit. For the Condition Register (CR), the field specified by **crf**D is affected by the ps0 computation for **ps_cmpo0** and **ps_cmpu0**, and by the ps1 computation for **ps_cmpo1** and **ps_cmpu1**. For all other paired single instructions, when RC=1, the CR1 field of the CR is set from FPSCR bits 0-3, which can be set by either the ps0 or the ps1 computation.

When in paired single mode (HID2[PSE] = 1), all the double-precision instructions are still valid, and execute as in non-paired single mode. In paired single mode, all the single-precision floating-point instructions (**fadds**, **fsubs**, **fmuls**, **fdivs**, **fmadds**, **fmsubs**, **fnmadds**, **fnmsubs, fres, frsp**) are valid, and operate on the ps0 operand (the double-precision operand, in the case of **frsp**) of the specified registers. The ps1 value in the destination register is duplicated from the ps0 result in such an operation. (See Page 12-85 for an exception about **frsp**.**)** The load floating-point single instructions (**lfs**[**u**][**x**]) load a single-precision floating-point value into the ps0 position of the FPR, and duplicate that value in the ps1 position. The store floating-point single instructions (**stfs**[**u**][**x**]) store the ps0 value only.

The relationship between the internal format for paired single operands and that for double-precision floating-point operands is unspecified. It is a programming error to apply double-precision instructions to paired single operands and vice versa. In particular, loading an operand as a double and then storing it as a paired single will not yield the original value back in memory. This presents a problem when it is desired to save the state of FPRs so that they can later be restored, particularly in the case of an interrupt.

The solution to this problem is that the following sequence of store and load instructions, executed

when HID2[PSE] = 1, is guaranteed to restore the state of floating-point register **fr**X regardless of its format. Assume GQR0 contains the value 0, indicating that no conversion takes place on paired single quantized loads and stores. Then save each register using the instruction pair:

psq_st **fr**X,0(**r**1),0,0
stfd **fr**X,8(**r**1)

and restore each register using the instruction pair:

psq_l **fr**X,0(**r**1),0,0
lfd **fr**X,8(**r**1)

Note that restoration of the ps1 value of a paired single operand is not exact in the following sense. If the ps1 value is a Denorm, it will get stored as the value 0, and so its restored value will also be the value 0.

Programming Note—Conversion from a double-precision operand to a single-precision operand when HID2[PSE] = 1 is accomplished using **frsp**, which takes a double-precision operand as input and produces a single-precision result in ps0 of the destination register. (See page 12-85.) Conversion from a single-precision operand to a double-precision operand, on the other hand, requires a software conversion routine, in general. However, the Gekko processor supports the following performance enhancement to implement this conversion. Any single-precision value in ps0 can be used as the input operand to a double-precision floating-point instruction, including a store.

Note that when HID2[PSE] = 1, the **fctiw** and **fctiwz** instructions give the expected result when used with the **stfiwx** instruction to store the resultant integer. Since these are are both classified as double-precision instructions, the integer result is placed in the low order word of the double-precision operand in the destination FPR. Like other double-precision results, these cannot then be operated on or stored using paired single operations.

Each of the paired single operands or result values behave the same way as single-precision operands or results in the following two tables. Table 2-21 summarizes the conditions and mode behavior for operands.

Table 2-21. Floating-Point Operand Data Type Behavior

Operand A Data Type	Operand B Data Type	Operand C Data Type	IEEE Mode (NI = 0)	Non-IEEE Mode (NI = 1)
Single denormalized Double denormalized	Single denormalized Double denormalized	Single denormalized Double denormalized	Normalize all three	Zero all three
Single denormalized Double denormalized	Single denormalized Double denormalized	Normalized or zero	Normalize A and B	Zero A and B
Normalized or zero	Single denormalized Double denormalized	Single denormalized Double denormalized	Normalize B and C	Zero B and C
Single denormalized Double denormalized	Normalized or zero	Single denormalized Double denormalized	Normalize A and C	Zero A and C
Single denormalized Double denormalized	Normalized or zero	Normalized or zero	Normalize A	Zero A
Normalized or zero	Single denormalized Double denormalized	Normalized or zero	Normalize B	Zero B
Normalized or zero	Normalized or zero	Single denormalized Double denormalized	Normalize C	Zero C

Table 2-21. Floating-Point Operand Data Type Behavior (Continued)

Operand A Data Type	Operand B Data Type	Operand C Data Type	IEEE Mode (NI = 0)	Non-IEEE Mode (NI = 1)
Single QNaN Single SNaN Double QNaN Double SNaN	Don't care	Don't care	QNaN[1]	QNaN[1]
Don't care	Single QNaN Single SNaN Double QNaN Double SNaN	Don't care	QNaN[1]	QNaN[1]
Don't care	Don't care	Single QNaN Single SNaN Double QNaN Double SNaN	QNaN[1]	QNaN[1]
Single normalized Single infinity Single zero Double normalized Double infinity Double zero	Single normalized Single infinity Single zero Double normalized Double infinity Double zero	Single normalized Single infinity Single zero Double normalized Double infinity Double zero	Do the operation	Do the operation

[1] Prioritize according to Chapter 3, "Operand Conventions," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Table 2-22 summarizes the mode behavior for results.

Table 2-22. Floating-Point Result Data Type Behavior

Precision	Data Type	IEEE Mode (NI = 0)	Non-IEEE Mode (NI = 1)
Single	Denormalized	Return single-precision denormalized number with trailing zeros.	Return zero.
Single	Normalized, infinity, zero	Return the result.	Return the result.
Single	QNaN, SNaN	Return QNaN.	Return QNaN.

Single	INT	Place integer into low word of FPR.	If (Invalid Operation)
			then
			Place (0x8000) into FPR[32–63]
			else
			Place integer into FPR[32–63].

Double	Denormalized	Return double-precision denormalized number.	Return zero.
Double	Normalized, infinity, zero	Return the result.	Return the result.
Double	QNaN, SNaN	Return QNaN.	Return QNaN.
Double	INT	Not supported by Gekko	Not supported by Gekko

2.3 Instruction Set Summary

This chapter describes instructions and addressing modes defined for Gekko. These instructions are divided into the following functional categories:

- Integer instructions—These include arithmetic and logical instructions. For more information, see Section 2.3.4.1, "Integer Instructions" on Page 2-37.
- Floating-point instructions—These include floating-point arithmetic instructions (single-precision, double-precision and paired single), as well as instructions that affect the floating-point status and control register (FPSCR). For more information, see Section 2.3.4.2, "Floating-Point Instructions" on Page 2-41.
- Load and store instructions—These include integer and floating-point (including quantized) load and store instructions. For more information, see Section 2.3.4.3, "Load and Store Instructions" on Page 2-46.
- Flow control instructions—These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow. For more information, see Section 2.3.4.4, "Branch and Flow Control Instructions" on Page 2-58.
- Processor control instructions—These instructions are used for synchronizing memory accesses and managing caches, TLBs, and segment registers. For more information, see Section 2.3.4.6, "Processor Control Instructions—UISA" on Page 2-61, Section 2.3.5.1, "Processor Control Instructions—VEA" on Page 2-65, and Section 2.3.6.2, "Processor Control Instructions—OEA" on Page 2-71.
- Memory synchronization instructions—These instructions are used for memory synchronizing. For more information, see Section 2.3.4.7, "Memory Synchronization Instructions—UISA" on Page 2-64 and Section 2.3.5.2, "Memory Synchronization Instructions—VEA" on Page 2-66.
- Memory control instructions—These instructions provide control of caches, TLBs, and segment registers. For more information, see Section 2.3.5.3, "Memory Control Instructions—VEA" on Page 2-67 and Section 2.3.6.3, "Memory Control Instructions—OEA" on Page 2-71.
- External control instructions—These include instructions for use with special input/output devices. For more information, see Section 2.3.5.4, "Optional External Control Instructions" on Page 2-69.

NOTE: This grouping of instructions does not necessarily indicate the execution unit that processes a particular instruction or group of instructions. That information, which is useful for scheduling instructions most effectively, is provided in Chapter 6, "Instruction Timing" in this manual.

Integer instructions operate on word operands. Floating-point instructions operate on single-precision, double-precision and paired single floating-point operands. The PowerPC architecture uses instructions that are four bytes long and word-aligned. It provides for byte, half-word, and word operand loads and stores between memory and a set of 32 general-purpose registers (GPRs). It provides for word and double-word operand loads and stores between memory and a set of 32 floating-point registers (FPRs). In addition, the Gekko implementation provides for byte, half word, word and double word quantized loads and stores between memory and the FPRs.

Arithmetic and logical instructions do not read or modify memory. To use the contents of a memory location in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written to the target location using load and store instructions.

The description of each instruction includes the mnemonic and a formatted list of operands. To

simplify assembly language programming, a set of simplified mnemonics and symbols is provided for some of the frequently-used instructions; see Appendix F, “Simplified Mnemonics,” in the *PowerPC Microprocessor Family: The Programming Environments* manual for a complete list of simplified mnemonics. Note that the architecture specification refers to simplified mnemonics as extended mnemonics. Programs written to be portable across the various assemblers for the PowerPC architecture should not assume the existence of mnemonics not described in that document.

2.3.1 Classes of Instructions

The Gekko instructions belong to one of the following three classes:

- Defined
- Illegal
- Reserved

Note that while the definitions of these terms are consistent among the PowerPC processors, the assignment of these classifications is not. For example, PowerPC instructions defined for 64-bit implementations are treated as illegal by 32-bit implementations such as Gekko.

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

Instruction encodings that are now illegal may become assigned to instructions in the architecture or may be reserved by being assigned to processor-specific instructions.

2.3.1.1 Definition of Boundedly Undefined

If instructions are encoded with incorrectly set bits in reserved fields, the results on execution can be said to be boundedly undefined. If a user-level program executes the incorrectly coded instruction, the resulting undefined results are bounded in that a spurious change from user to supervisor state is not allowed, and the level of privilege exercised by the program in relation to memory access and other system resources cannot be exceeded. Boundedly-undefined results for a given instruction may vary between implementations, and between execution attempts in the same implementation.

2.3.1.2 Defined Instruction Class

Defined instructions are guaranteed to be supported in all PowerPC implementations, except as stated in the instruction descriptions in Chapter 12, "Instruction Set" in this manual. Gekko provides hardware support for all instructions defined for 32-bit implementations.

It does not support the optional **fsqrt**, **fsqrts**, and **tlbia** instructions.

A PowerPC processor invokes the illegal instruction error handler (part of the program exception) when the unimplemented PowerPC instructions are encountered so they may be emulated in software, as required. Note that the architecture specification refers to exceptions as interrupts.

A defined instruction can have invalid forms. Gekko provides limited support for instructions represented in an invalid form.

2.3.1.3 Illegal Instruction Class

Illegal instructions can be grouped into the following categories:

- Instructions not defined in the PowerPC architecture.The following primary opcodes are defined as illegal but may be used in future extensions to the architecture:1, 5, 6, 9, 22

  Future versions of the PowerPC architecture may define any of these instructions to perform new functions.

- Instructions defined in the PowerPC architecture but not implemented in a specific PowerPC implementation. For example, instructions that can be executed on 64-bit PowerPC processors are considered illegal by 32-bit processors such as Gekko.

  The following primary opcodes are defined for 64-bit implementations only and are illegal on Gekko:2, 30, 58, 62

- All unused extended opcodes are illegal. The unused extended opcodes can be determined from information in Section A.1 and Section 2.3.1.4, "Reserved Instruction Class" on Page 2-34. Notice that extended opcodes for instructions defined only for 64-bit implementations are illegal in 32-bit implementations, and vice versa.
  The following primary opcodes have unused extended opcodes.

  4, 17, 19, 31, 59, 63 (Primary opcodes 30 and 62 are illegal for all 32-bit implementations, but as 64-bit opcodes they have some unused extended opcodes.)

- An instruction consisting of only zeros is guaranteed to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized memory invokes the system illegal instruction error handler (a program exception). Note that if only the primary opcode consists of all zeros, the instruction is considered a reserved instruction, as described in Section 2.3.1.4.

Gekko invokes the system illegal instruction error handler (a program exception) when it detects any instruction from this class or any instructions defined only for 64-bit implementations.

See Section 4.5.7, "Program Exception (0x00700)" on Page 4-19 for additional information about illegal and invalid instruction exceptions. Except for an instruction consisting of binary zeros, illegal instructions are available for additions to the PowerPC architecture.

2.3.1.4 Reserved Instruction Class

Reserved instructions are allocated to specific implementation-dependent purposes not defined by the PowerPC architecture. Attempting to execute an unimplemented reserved instruction invokes the illegal instruction error handler (a program exception). See Section 4.5.7, "Program Exception (0x00700)" on Page 4-19 for information about illegal and invalid instruction exceptions.

The PowerPC architecture defines four types of reserved instructions:

- Instructions in the POWER architecture not part of the PowerPC UISA. For details on POWER architecture incompatibilities and how they are handled by PowerPC processors, see Appendix B, “POWER Architecture Cross Reference" in the *PowerPC Microprocessor Family: The Programming Environments* manual.
- Implementation-specific instructions required for the processor to conform to the PowerPC architecture (none of these are implemented in Gekko)
- All other implementation-specific instructions
- Architecturally-allowed extended opcodes

2.3.1.5 Gekko’s implementation-specific instructions

The Gekko processor includes extensions to the PowerPC architecture to enhance the performance of graphics applications. The new instructions include a new cache control instruction, **dcbz_l**, four quantized load and four quantized store instructions, and 29 paired single floating-point instructions. These new instructions are implemented using primary opcodes 4, 56, 57, 60 and 61. See Chapter 9 for a description of the graphics enhancement features and Chapter 12, "Instruction Set" in this manual for a detailed description of the new instructions.

2.3.2 Addressing Modes

This section provides an overview of conventions for addressing memory and for calculating effective addresses as defined by the PowerPC architecture for 32-bit implementations. For more detailed information, see “Conventions” in Chapter 4, “Addressing Modes and Instruction Set Summary" of the *PowerPC Microprocessor Family: The Programming Environments* manual.

2.3.2.1 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a memory access or branch instruction or when it fetches the next sequential instruction.

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

2.3.2.2 Memory Operands

Memory operands may be bytes, half words, words, or double words, or, for the load/store multiple and load/store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction. The PowerPC architecture supports both big-endian and little-endian byte ordering. The default byte and bit ordering is big-endian. See “Byte Ordering" in Chapter 3, “Operand Conventions” of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information about big- and little-endian byte ordering.

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length. In other words, the “natural” address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is misaligned.

For a detailed discussion about memory operands, see Chapter 3, “Operand Conventions” of the *PowerPC Microprocessor Family: The Programming Environments* manual.

2.3.2.3 Effective Address Calculation

An effective address is the 32-bit sum computed by the processor when executing a memory access or branch instruction or when fetching the next sequential instruction. For a memory access instruction, if the sum of the effective address and the operand length exceeds the maximum effective address, the memory operand is considered to wrap around from the maximum effective address through effective address 0, as described in the following paragraphs.

Effective address computations for both data and instruction accesses use 32-bit unsigned binary arithmetic. A carry from bit 0 is ignored.

Load and store operations have the following modes of effective address generation:

- EA = (**r**A|0) + offset (including offset = 0) (register indirect with immediate index)
- EA = (**r**A|0) + **r**B (register indirect with index)

Refer to Section 2.3.4.3.2, "Integer Load and Store Address Generation" on Page 2-47 for a detailed description of effective address generation for load and store operations.

Branch instructions have three categories of effective address generation:

- Immediate
- Link register indirect
- Count register indirect

2.3.2.4 Synchronization

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

2.3.2.4.1 Context Synchronization

The System Call (**sc**) and Return from Interrupt (**rfi**) instructions perform context synchronization by allowing previously issued instructions to complete before performing a change in context. Execution of one of these instructions ensures the following:

- No higher priority exception exists (**sc**).
- All previous instructions have completed to a point where they can no longer cause an exception. If a prior memory access instruction causes direct-store error exceptions, the results are guaranteed to be determined before this instruction is executed.
- Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
- The instructions following the **sc** or **rfi** instruction execute in the context established by these instructions.

2.3.2.4.2 Execution Synchronization

An instruction is execution synchronizing if all previously initiated instructions appear to have completed before the instruction is initiated or, in the case of **sync** and **isync**, before the instruction completes. For example, the Move to Machine State Register (**mtmsr**) instruction is execution synchronizing. It ensures that all preceding instructions have completed execution and cannot cause an exception before the instruction executes, but does not ensure subsequent instructions execute in the newly established environment. For example, if the **mtmsr** sets the MSR[PR] bit, unless an **isync** immediately follows the **mtmsr** instruction, a privileged instruction could be executed or privileged access could be performed without causing an exception even though the MSR[PR] bit indicates user mode.

2.3.2.4.3 Instruction-Related Exceptions

There are two kinds of exceptions in Gekko—those caused directly by the execution of an instruction and those caused by an asynchronous event (or interrupts). Either may cause components of the system software to be invoked.

Exceptions can be caused directly by the execution of an instruction as follows:

- An attempt to execute an illegal instruction causes the illegal instruction (program exception) handler to be invoked. Note that the **dcbz_l** instruction is illegal when HID2[LCE] = 0, the **psq_l**, **psq_lu**, **psq_st** and **psq_stu** instructions are illegal when HID2[PSQE] = 0 or HID2[PSE] = 0, and all other paired single instructions are illegal when HID2[PSE] = 0. An attempt by a user-level program to execute the supervisor-level instructions listed below causes the privileged instruction (program exception) handler to be invoked. Gekko provides the following supervisor-level instructions: **dcbi**, **mfmsr**, **mfspr**, **mfsr**, **mfsrin**, **mtmsr**, **mtspr**, **mtsr**, **mtsrin**, **rfi**, **tlbie**, and **tlbsync**. Note that the privilege level of the **mfspr** and **mtspr** instructions depends on the SPR encoding.
- Any **mtspr**, **mfspr**, or **mftb** instruction with an invalid SPR (or TBR) field causes an illegal type program exception. Likewise, a program exception is taken if user-level software tries to access a supervisor-level SPR. An **mtspr** instruction executing in supervisor mode (MSR[PR] = 0) with the SPR field specifying HID1 or PVR (read-only registers) executes as a no-op.

- An attempt to access memory that is not available (page fault) causes the ISI or DSI exception handler to be invoked.
- The execution of an **sc** instruction invokes the system call exception handler that permits a program to request the system to perform a service.
- The execution of a trap instruction invokes the program exception trap handler.
- The execution of an instruction that causes a floating-point exception while exceptions are enabled in the MSR invokes the program exception handler.

A detailed description of exception conditions is provided in Chapter 4, "Exceptions" in this manual.

2.3.3 Instruction Set Overview

This section provides a brief overview of the PowerPC instructions implemented in Gekko and highlights any special information with respect to how Gekko implements a particular instruction. Note that the categories used in this section correspond to those used in Chapter 4, “Addressing Modes and Instruction Set Summary” in the *PowerPC Microprocessor Family: The Programming Environments* manual. These categorizations are somewhat arbitrary and are provided for the convenience of the programmer and do not necessarily reflect the PowerPC architecture specification.

Note that some instructions have the following optional features:

- CR Update—The dot (**.**) suffix on the mnemonic enables the update of the CR.
- Overflow option—The **o** suffix indicates that the overflow bit in the XER is enabled.

2.3.4 PowerPC UISA Instructions

The PowerPC UISA includes the base user-level instruction set (excluding a few user-level cache control, synchronization, and time base instructions), user-level registers, programming model, data types, and addressing modes. This section discusses the instructions defined in the UISA.

2.3.4.1 Integer Instructions

This section describes the integer instructions. These consist of the following:

- Integer arithmetic instructions
- Integer compare instructions
- Integer logical instructions
- Integer rotate and shift instructions

Integer instructions use the content of the GPRs as source operands and place results into GPRs, into the integer exception register (XER), and into condition register (CR) fields.

2.3.4.1.1 Integer Arithmetic Instructions

Table 2-23 lists the integer arithmetic instructions for the PowerPC processors.

Table 2-23. Integer Arithmetic Instructions

Name	Mnemonic	Syntax
Add Immediate	**addi**	**r**D**,r**A,SIMM
Add Immediate Shifted	**addis**	**r**D**,r**A,SIMM
Add	**add** (**add. addo addo.**)	**r**D**,r**A**,r**B

Table 2-23. Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Syntax
Subtract From	**subf** (**subf. subfo subfo.**)	**r**D,**r**A,**r**B
Add Immediate Carrying	**addic**	**r**D,**r**A,SIMM
Add Immediate Carrying and Record	**addic.**	**r**D,**r**A,SIMM
Subtract from Immediate Carrying	**subfic**	**r**D,**r**A,SIMM
Add Carrying	**addc** (**addc. addco addco.**)	**r**D,**r**A,**r**B
Subtract from Carrying	**subfc** (**subfc. subfco subfco.**)	**r**D,**r**A,**r**B
Add Extended	**adde** (**adde. addeo addeo.**)	**r**D,**r**A,**r**B
Subtract from Extended	**subfe** (**subfe. subfeo subfeo.**)	**r**D,**r**A,**r**B
Add to Minus One Extended	**addme** (**addme. addmeo addmeo.**)	**r**D,**r**A
Subtract from Minus One Extended	**subfme** (**subfme. subfmeo subfmeo.**)	**r**D,**r**A
Add to Zero Extended	**addze** (**addze. addzeo addzeo.**)	**r**D,**r**A
Subtract from Zero Extended	**subfze** (**subfze. subfzeo subfzeo.**)	**r**D,**r**A
Negate	**neg** (**neg. nego nego.**)	**r**D,**r**A
Multiply Low Immediate	**mulli**	**r**D,**r**A,SIMM
Multiply Low	**mullw** (**mullw. mullwo mullwo.**)	**r**D,**r**A,**r**B
Multiply High Word	**mulhw** (**mulhw.**)	**r**D,**r**A,**r**B
Multiply High Word Unsigned	**mulhwu** (**mulhwu.**)	**r**D,**r**A,**r**B
Divide Word	**divw** (**divw. divwo divwo.**)	**r**D,**r**A,**r**B
Divide Word Unsigned	**divwu divwu. divwuo divwuo.**	**r**D,**r**A,**r**B

Although there is no Subtract Immediate instruction, its effect can be achieved by using an **addi** instruction with the immediate operand negated. Simplified mnemonics are provided that include this negation. The **subf** instructions subtract the second operand (**r**A) from the third operand (**r**B). Simplified mnemonics are provided in which the third operand is subtracted from the second operand. See Appendix F, “Simplified Mnemonics,” in the *PowerPC Microprocessor Family: The Programming Environments* manual for examples.

The UISA states that an implementation that executes instructions that set the overflow enable bit (OE) or the carry bit (CA) may either execute these instructions slowly or prevent execution of the subsequent instruction until the operation completes. Chapter 6 describes how Gekko handles CR dependencies. The summary overflow bit (SO) and overflow bit (OV) in the integer exception register are set to reflect an overflow condition of a 32-bit result. This can happen only when OE = 1.

2.3.4.1.2 Integer Compare Instructions

The integer compare instructions algebraically or logically compare the contents of register **r**A with either the zero-extended value of the UIMM operand, the sign-extended value of the SIMM operand, or the contents of register **r**B. The comparison is signed for the **cmpi** and **cmp** instructions, and unsigned for the **cmpli** and **cmpl** instructions.

Table 2-24 summarizes the integer compare instructions.

Table 2-24. Integer Compare Instructions

Name	Mnemonic	Syntax
Compare Immediate	**cmpi**	**crf**D,L,**r**A,SIMM
Compare	**cmp**	**crf**D,L,**r**A,**r**B
Compare Logical Immediate	**cmpli**	**crf**D,L,**r**A,UIMM
Compare Logical	**cmpl**	**crf**D,L,**r**A,**r**B

The **crf**D operand can be omitted if the result of the comparison is to be placed in CR0. Otherwise the target CR field must be specified in **crf**D, using an explicit field number.

For information on simplified mnemonics for the integer compare instructions see Appendix F, "Simplified Mnemonics," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

2.3.4.1.3 Integer Logical Instructions

The logical instructions shown in Table 2-25 perform bit-parallel operations on the specified operands. Logical instructions with the CR updating enabled (uses dot suffix) and instructions **andi.** and **andis.** set CR field CR0 to characterize the result of the logical operation. Logical instructions do not affect XER[SO], XER[OV], or XER[CA].

See Appendix F, "Simplified Mnemonics," in the *PowerPC Microprocessor Family: The Programming Environments* manual for simplified mnemonic examples for integer logical operations.

Table 2-25. Integer Logical Instructions

Name	Mnemonic	Syntax	Implementation Notes
AND Immediate	**andi.**	**r**A,**r**S,UIMM	—
AND Immediate Shifted	**andis.**	**r**A,**r**S,UIMM	—
OR Immediate	**ori**	**r**A,**r**S,UIMM	The PowerPC architecture defines **ori r0,r0,0** as the preferred form for the no-op instruction. The dispatcher discards this instruction (except for pending trace or breakpoint exceptions).
OR Immediate Shifted	**oris**	**r**A,**r**S,UIMM	—
XOR Immediate	**xori**	**r**A,**r**S,UIMM	—
XOR Immediate Shifted	**xoris**	**r**A,**r**S,UIMM	—
AND	**and** (**and.**)	**r**A,**r**S,**r**B	—
OR	**or** (**or.**)	**r**A,**r**S,**r**B	—

Table 2-25. Integer Logical Instructions (Continued)

Name	Mnemonic	Syntax	Implementation Notes
XOR	**xor** (**xor.**)	**r**A,**r**S,**r**B	—
NAND	**nand** (**nand.**)	**r**A,**r**S,**r**B	—
NOR	**nor** (**nor.**)	**r**A,**r**S,**r**B	—
Equivalent	**eqv** (**eqv.**)	**r**A,**r**S,**r**B	—
AND with Complement	**andc** (**andc.**)	**r**A,**r**S,**r**B	—
OR with Complement	**orc** (**orc.**)	**r**A,**r**S,**r**B	—
Extend Sign Byte	**extsb** (**extsb.**)	**r**A,**r**S	—
Extend Sign Half Word	**extsh** (**extsh.**)	**r**A,**r**S	—
Count Leading Zeros Word	**cntlzw** (**cntlzw.**)	**r**A,**r**S	—

2.3.4.1.4 Integer Rotate Instructions

Rotation operations are performed on data from a GPR, and the result, or a portion of the result, is returned to a GPR. See Appendix F, “Simplified Mnemonics,” in the *PowerPC Microprocessor Family: The Programming Environments* manual for a complete list of simplified mnemonics that allows simpler coding of often-used functions such as clearing the leftmost or rightmost bits of a register, left justifying or right justifying an arbitrary field, and simple rotates and shifts.

Integer rotate instructions rotate the contents of a register. The result of the rotation is either inserted into the target register under control of a mask (if a mask bit is 1 the associated bit of the rotated data is placed into the target register, and if the mask bit is 0 the associated bit in the target register is unchanged), or ANDed with a mask before being placed into the target register.

The integer rotate instructions are summarized in Table 2-26.

Table 2-26. Integer Rotate Instructions

Name	Mnemonic	Syntax
Rotate Left Word Immediate then AND with Mask	**rlwinm** (**rlwinm.**)	**r**A,**r**S,SH,MB,ME
Rotate Left Word then AND with Mask	**rlwnm** (**rlwnm.**)	**r**A,**r**S,**r**B,MB,ME
Rotate Left Word Immediate then Mask Insert	**rlwimi** (**rlwimi.**)	**r**A,**r**S,SH,MB,ME

2.3.4.1.5 Integer Shift Instructions

The integer shift instructions perform left and right shifts. Immediate-form logical (unsigned) shift operations are obtained by specifying masks and shift values for certain rotate instructions. Simplified mnemonics (shown in Appendix F, “Simplified Mnemonics,” in the *PowerPC Microprocessor Family: The Programming Environments* manual) are provided to make coding of such shifts simpler and easier to understand.

Multiple-precision shifts can be programmed as shown in Appendix C, “Multiple-Precision Shifts" in the *PowerPC Microprocessor Family: The Programming Environments* manual. The integer shift instructions are summarized in Table 2-27.

Table 2-27. Integer Shift Instructions

Name	Mnemonic	Syntax
Shift Left Word	**slw** (**slw.**)	**r**A,**r**S,**r**B
Shift Right Word	**srw** (**srw.**)	**r**A,**r**S,**r**B
Shift Right Algebraic Word Immediate	**srawi** (**srawi.**)	**r**A,**r**S,SH
Shift Right Algebraic Word	**sraw** (**sraw.**)	**r**A,**r**S,**r**B

2.3.4.2 Floating-Point Instructions

This section describes the floating-point instructions, which include the following:

- Floating-point arithmetic instructions
- Floating-point multiply-add instructions
- Floating-point rounding and conversion instructions
- Floating-point compare instructions
- Floating-point status and control register instructions
- Floating-point move instructions

See Section 2.3.4.3, "Load and Store Instructions" on Page 2-46 for information about floating-point loads and stores.

The PowerPC architecture supports a floating-point system as defined in the IEEE 754 standard, but requires software support to conform with that standard. All floating-point operations conform to the IEEE 754 standard, except if software sets the non-IEEE mode FPSCR[NI].

2.3.4.2.1 Floating-Point Arithmetic Instructions

The floating-point arithmetic instructions are summarized in Table 2-28.

Table 2-28. Floating-Point Arithmetic Instructions

Name	Mnemonic	Syntax
Floating Add (Double-Precision)	**fadd** (**fadd.**)	**fr**D,**fr**A,**fr**B
Floating Add Single	**fadds** (**fadds.**)	**fr**D,**fr**A,**fr**B
Floating Subtract (Double-Precision)	**fsub** (**fsub.**)	**fr**D,**fr**A,**fr**B
Floating Subtract Single	**fsubs** (**fsubs.**)	**fr**D,**fr**A,**fr**B
Floating Multiply (Double-Precision)	**fmul** (**fmul.**)	**fr**D,**fr**A,**fr**C
Floating Multiply Single	**fmuls** (**fmuls.**)	**fr**D,**fr**A,**fr**C
Floating Divide (Double-Precision)	**fdiv** (**fdiv.**)	**fr**D,**fr**A,**fr**B
Floating Divide Single	**fdivs** (**fdivs.**)	**fr**D,**fr**A,**fr**B
Floating Reciprocal Estimate Single [1]	**fres** (**fres.**)	**fr**D,**fr**B
Floating Reciprocal Square Root Estimate [1]	**frsqrte** (**frsqrte.**)	**fr**D,**fr**B
Floating Select [1]	**fsel** (**fsel.**)	**fr**D,**fr**A,**fr**C,**fr**B
Paired Single Add [2]	**ps_add** (**ps_add.**)	**fr**D,**fr**A,**fr**B
Paired Single Subtract [2]	**ps_sub** (**ps_sub.**)	**fr**D,**fr**A,**fr**B
Paired Single Multiply [2]	**ps_mul** (**ps_mul.**)	**fr**D,**fr**A,**fr**C
Paired Single Divide [2]	**ps_div** (**ps_div.**)	**fr**D,**fr**A,**fr**B
Paired Single Reciprocal Estimate [2]	**ps_res** (**ps_res.**)	**fr**D,**fr**B
Paired Single Reciprocal Square Root Estimate [2]	**ps_rsqrte** (**ps_rsqrte.**)	**fr**D,**fr**B
Paired Single Select [2]	**ps_sel** (**ps_sel.**)	**fr**D,**fr**A,**fr**C,**fr**B
Paired Single Multiply Scalar High [2]	**ps_muls0** (**ps_muls0.**)	**fr**D,**fr**A,**fr**C
Paired Single Multiply Scalar Low [2]	**ps_muls1** (**ps_muls1.**)	**fr**D,**fr**A,**fr**C
Paired Single Vector Sum High [2]	**ps_sum0** (**ps_sum0.**)	**fr**D,**fr**A,**fr**C,**fr**B
Paired Single Vector Sum Low [2]	**ps_sum1** (**ps_sum1.**)	**fr**D,**fr**A,**fr**C,**fr**B

Note: [1]The **fres**, **frsqrte** and **fsel** instructions are optional in the PowerPC architecture.
Note: [2]These instructions belong to the Gekko graphics extensions, and are legal only when HID2[PSE] = 1.

Double-precision arithmetic instructions, except those involving multiplication (**fmul**, **fmadd**, **fmsub**, **fnmadd**, **fnmsub**) execute with the same latency as their single-precision equivalents. For additional details on floating-point performance, refer to Chapter 6, "Instruction Timing" in this manual.

2.3.4.2.2 Floating-Point Multiply-Add Instructions

These instructions combine multiply and add operations without an intermediate rounding operation. The floating-point multiply-add instructions are summarized in Table 2-29.

Table 2-29. Floating-Point Multiply-Add Instructions

Name	Mnemonic	Syntax
Floating Multiply-Add (Double-Precision)	**fmadd** (**fmadd.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Multiply-Add Single	**fmadds** (**fmadds.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Multiply-Subtract (Double-Precision)	**fmsub** (**fmsub.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Multiply-Subtract Single	**fmsubs** (**fmsubs.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Negative Multiply-Add (Double-Precision)	**fnmadd** (**fnmadd.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Negative Multiply-Add Single	**fnmadds** (**fnmadds.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Negative Multiply-Subtract (Double-Precision)	**fnmsub** (**fnmsub.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Negative Multiply-Subtract Single	**fnmsubs** (**fnmsubs.**)	**fr**D,**fr**A,**fr**C,**fr**B
Paired Single Multiply-Add [1]	**ps_madd** (**ps_madd.**)	**fr**D,**fr**A,**fr**C,**fr**B
Paired Single Multiply-Subtract [1]	**ps_msub** (**ps_msub.**)	**fr**D,**fr**A,**fr**C,**fr**B
Paired Single Negative Multiply-Add [1]	**ps_nmadd** (**ps_nmadd.**)	**fr**D,**fr**A,**fr**C,**fr**B
Paired Single Negative Multiply-Subtract [1]	**ps_nmsub** (**ps_nmsub.**)	**fr**D,**fr**A,**fr**C,**fr**B
Paired Single Multiply-Add Scalar High [1]	**ps_madds0** (**ps_madds0.**)	**fr**D,**fr**A,**fr**C,**fr**B
Paired Single Multiply-Add Scalar Low [1]	**ps_madds1** (**ps_madds1.**)	**fr**D,**fr**A,**fr**C,**fr**B

Note: [1]These instructions are Gekko-specific, and are legal only when HID2[PSE] = 1.

2.3.4.2.3 Floating-Point Rounding and Conversion Instructions

The Floating Round to Single-Precision (**frsp**) instruction is used to truncate a 64-bit double-precision number to a 32-bit single-precision floating-point number. The floating-point convert instructions convert a 64-bit double-precision floating-point number to a 32-bit signed integer number.

Examples of uses of these instructions to perform various conversions can be found in Appendix D, “Floating-Point Models,” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Table 2-30. Floating-Point Rounding and Conversion Instructions

Name	Mnemonic	Syntax
Floating Round to Single	**frsp** (**frsp.**)	**fr**D,**fr**B
Floating Convert to Integer Word	**fctiw** (**fctiw.**)	**fr**D,**fr**B
Floating Convert to Integer Word with Round toward Zero	**fctiwz** (**fctiwz.**)	**fr**D,**fr**B

2.3.4.2.4 Floating-Point Compare Instructions

Floating-point compare instructions compare the contents of two floating-point registers. The comparison ignores the sign of zero (that is +0 = –0).

The floating-point compare instructions are summarized in Table 2-31.

Table 2-31. Floating-Point Compare Instructions

Name	Mnemonic	Syntax
Floating Compare Unordered	**fcmpu**	**crf**D,**fr**A,**fr**B
Floating Compare Ordered	**fcmpo**	**crf**D,**fr**A,**fr**B
Paired Single Compare Unordered High [1]	**ps_cmpu0**	**crf**D,**fr**A,**fr**B
Paired Single Compare Unordered Low [1]	**ps_cmpu1**	**crf**D,**fr**A,**fr**B
Paired Single Compare Ordered High [1]	**ps_cmpo0**	**crf**D,**fr**A,**fr**B
Paired Single Compare Ordered Low [1]	**ps_cmpo1**	**crf**D,**fr**A,**fr**B

Note: [1]These instructions are Gekko-specific, and are legal only when HID2[PSE] = 1.

The PowerPC architecture allows an **fcmpu** or **fcmpo** instruction with the Rc bit set to produce a boundedly-undefined result, which may include an illegal instruction program exception. In Gekko, **crf**D should be treated as undefined

2.3.4.2.5 Floating-Point Status and Control Register Instructions

Every FPSCR instruction appears to synchronize the effects of all floating-point instructions executed by a given processor. Executing an FPSCR instruction ensures that all floating-point instructions previously initiated by the given processor appear to have completed before the FPSCR instruction is initiated and that no subsequent floating-point instructions appear to be initiated by the given processor until the FPSCR instruction has completed.

The FPSCR instructions are summarized in Table 2-32.

Table 2-32. Floating-Point Status and Control Register Instructions

Name	Mnemonic	Syntax
Move from FPSCR	**mffs** (**mffs.**)	**fr**D
Move to Condition Register from FPSCR	**mcrfs**	**crf**D,**crf**S
Move to FPSCR Field Immediate	**mtfsfi** (**mtfsfi.**)	**crf**D,IMM
Move to FPSCR Fields	**mtfsf** (**mtfsf.**)	FM,**fr**B
Move to FPSCR Bit 0	**mtfsb0** (**mtfsb0.**)	**crb**D
Move to FPSCR Bit 1	**mtfsb1** (**mtfsb1.**)	**crb**D

Implementation Note—The PowerPC architecture states that in some implementations, the Move to FPSCR Fields (**mtfsf**) instruction may perform more slowly when only some of the fields are updated as opposed to all of the fields. In Gekko, there is no degradation of performance.

2.3.4.2.6 Floating-Point Move Instructions

Floating-point move instructions copy data from one FPR to another. The floating-point move instructions do not modify the FPSCR. The CR update option in these instructions controls the placing of result status into CR1.

Table 2-33 summarizes the floating-point move instructions.

Table 2-33. Floating-Point Move Instructions

Name	Mnemonic	Syntax
Floating Move Register	**fmr** (**fmr.**)	**fr**D,**fr**B
Floating Negate	**fneg** (**fneg.**)	**fr**D,**fr**B
Floating Absolute Value	**fabs** (**fabs.**)	**fr**D,**fr**B
Floating Negative Absolute Value	**fnabs** (**fnabs.**)	**fr**D,**fr**B
Paired Single Move Register [1]	**ps_mr** (**ps_mr.**)	**fr**D,**fr**B
Paired Single Negate [1]	**ps_neg** (**ps_neg.**)	**fr**D,**fr**B
Paired Single Absolute Value [1]	**ps_abs** (**ps_abs.**)	**fr**D,**fr**B
Paired Single Negative Absolute Value [1]	**ps_nabs** (**ps_nabs.**)	**fr**D,**fr**B
Paired Single Merge High [1]	**ps_merge00** (**ps_merge00.**)	**fr**D,**fr**A,**fr**B
Paired Single Merge Direct [1]	**ps_merge01** (**ps_merge01.**)	**fr**D,**fr**A,**fr**B
Paired Single Merge Swapped [1]	**ps_merge10** (**ps_merge10.**)	**fr**D,**fr**A,**fr**B
Paired Single Merge Low [1]	**ps_merge11** (**ps_merge11.**)	**fr**D,**fr**A,**fr**B

Note: [1]These instructions belong to the Gekko graphics extensions, and are legal only when HID2[PSE] = 1.

2.3.4.3 Load and Store Instructions

Load and store instructions are issued and translated in program order; however, the accesses can occur out of order. Synchronizing instructions are provided to enforce strict ordering. This section describes the load and store instructions, which consist of the following:

- Integer load instructions
- Integer store instructions
- Integer load and store with byte-reverse instructions
- Integer load and store multiple instructions
- Floating-point load instructions, including quantized loads
- Floating-point store instructions, including quantized stores
- Memory synchronization instructions

Implementation Notes—The following describes how Gekko handles misalignment:

Gekko provides hardware support for misaligned memory accesses. It performs those accesses within a single cycle if the operand lies within a double-word boundary. Misaligned memory accesses that cross a double-word boundary degrade performance.

For string operations, the hardware makes no attempt to combine register values to reduce the number

of discrete accesses. Combining stores enhances performance if store gathering is enabled and the accesses meet the criteria described in Section 6.4.7, "Integer Store Gathering" on Page 6-25. Note that the PowerPC architecture requires load/store multiple instruction accesses to be aligned. At a minimum, additional cache access cycles are required.

Although many unaligned memory accesses are supported in hardware, the frequent use of them is discouraged since they can compromise the overall performance of the processor.

Accesses that cross a translation boundary may be restarted. That is, a misaligned access that crosses a page boundary is completely restarted if the second portion of the access causes a page fault. This may cause the first access to be repeated.

On some processors, such as the 603, a TLB reload would cause an instruction restart. On Gekko, TLB reloads are done transparently and only a page fault causes a restart.

2.3.4.3.1 Self-Modifying Code

When a processor modifies a memory location that may be contained in the instruction cache, software must ensure that memory updates are visible to the instruction fetching mechanism. This can be achieved by the following instruction sequence:

```
dcbst    ! update memory
sync     ! wait for update
icbi     ! remove (invalidate) copy in instruction cache
isync    ! remove copy in own instruction buffer
```

These operations are required because the data cache is a write-back cache. Since instruction fetching bypasses the data cache, changes to items in the data cache may not be reflected in memory until the fetch operations complete.

Special care must be taken to avoid coherency paradoxes in systems that implement unified secondary caches, and designers should carefully follow the guidelines for maintaining cache coherency that are provided in the VEA, and discussed in Chapter 5, “Cache Model and Memory Coherency" in the *PowerPC Microprocessor Family: The Programming Environments* manual. Because Gekko does not broadcast the M bit for instruction fetches, external caches are subject to coherency paradoxes.

2.3.4.3.2 Integer Load and Store Address Generation

Integer load and store operations generate effective addresses using register indirect with immediate index mode, register indirect with index mode, or register indirect mode. See Section 2.3.2.3, "Effective Address Calculation" on Page 2-35 for information about calculating effective addresses. Note that in some implementations, operations that are not naturally aligned may suffer performance degradation. Refer to Section 4.5.6, "Alignment Exception (0x00600)" on Page 4-19 for additional information about load and store address alignment exceptions.

2.3.4.3.3 Integer Load Instructions

For integer load instructions, the byte, half word, or word addressed by the EA (effective address) is loaded into **r**D. Many integer load instructions have an update form, in which **r**A is updated with the generated effective address. For these forms, if **r**A ≠ 0 and **r**A ≠ **r**D (otherwise invalid), the EA is placed into **r**A and the memory element (byte, half word, or word) addressed by the EA is loaded into **r**D. Note that the PowerPC architecture defines load with update instructions with operand **r**A = 0 or **r**A = **r**D as invalid forms.

Table 2-34 summarizes the integer load instructions.

Table 2-34. Integer Load Instructions

Name	Mnemonic	Syntax
Load Byte and Zero	**lbz**	**r**D,d(**r**A**)**
Load Byte and Zero Indexed	**lbzx**	**r**D,**r**A,**r**B
Load Byte and Zero with Update	**lbzu**	**r**D,d(**r**A**)**
Load Byte and Zero with Update Indexed	**lbzux**	**r**D,**r**A,**r**B
Load Half Word and Zero	**lhz**	**r**D,d(**r**A**)**
Load Half Word and Zero Indexed	**lhzx**	**r**D,**r**A,**r**B
Load Half Word and Zero with Update	**lhzu**	**r**D,d(**r**A**)**
Load Half Word and Zero with Update Indexed	**lhzux**	**r**D,**r**A,**r**B
Load Half Word Algebraic	**lha**	**r**D,d(**r**A**)**
Load Half Word Algebraic Indexed	**lhax**	**r**D,**r**A,**r**B
Load Half Word Algebraic with Update	**lhau**	**r**D,d(**r**A**)**
Load Half Word Algebraic with Update Indexed	**lhaux**	**r**D,**r**A,**r**B
Load Word and Zero	**lwz**	**r**D,d(**r**A**)**
Load Word and Zero Indexed	**lwzx**	**r**D,**r**A,**r**B
Load Word and Zero with Update	**lwzu**	**r**D,d(**r**A**)**
Load Word and Zero with Update Indexed	**lwzux**	**r**D,**r**A,**r**B

Implementation Notes—The following notes describe the Gekko implementation of integer load instructions:

- The PowerPC architecture cautions programmers that some implementations of the architecture may execute the load half algebraic (**lha**, **lhax**) instructions with greater latency than other types of load instructions. This is not the case for Gekko; these instructions operate with the same latency as other load instructions.

- The PowerPC architecture cautions programmers that some implementations of the architecture may run the load/store byte-reverse (**lhbrx**, **lbrx**, **sthbrx**, **stwbrx**) instructions with greater latency than other types of load/store instructions. This is not the case for Gekko. These instructions operate with the same latency as the other load/store instructions.

- The PowerPC architecture describes some preferred instruction forms for load and store multiple instructions and integer move assist instructions that may perform better than other forms in some implementations. None of these preferred forms affect instruction performance on Gekko.
- The PowerPC architecture defines the **lwarx** and **stwcx.** as a way to update memory atomically. In Gekko, reservations are made on behalf of aligned 32-byte sections of the memory address space. Executing **lwarx** and **stwcx.** to a page marked write-through does not cause a DSI exception if the W bit is set, but as with other memory accesses, DSI exceptions can result for other reasons such as protection violations or page faults.
- In general, because **stwcx.** always causes an external bus transaction it has slightly worse performance characteristics than normal store operations.

2.3.4.3.4 Integer Store Instructions

For integer store instructions, the contents of **r**S are stored into the byte, half word or word in memory addressed by the EA (effective address). Many store instructions have an update form, in which **r**A is updated with the EA. For these forms, the following rules apply:

- If **r**A ≠ 0, the effective address is placed into **r**A.
- If **r**S = **r**A, the contents of register **r**S are copied to the target memory element, then the generated EA is placed into **r**A (**r**S).

The PowerPC architecture defines store with update instructions with **r**A = 0 as an invalid form. In addition, it defines integer store instructions with the CR update option enabled (Rc field, bit 31, in the instruction encoding = 1) to be an invalid form.

Table 2-35 summarizes the integer store instructions.

Table 2-35. Integer Store Instructions

Name	Mnemonic	Syntax
Store Byte	**stb**	**r**S,**d(r**A**)**
Store Byte Indexed	**stbx**	**r**S,**r**A,**r**B
Store Byte with Update	**stbu**	**r**S,**d(r**A**)**
Store Byte with Update Indexed	**stbux**	**r**S,**r**A,**r**B
Store Half Word	**sth**	**r**S,**d(r**A**)**
Store Half Word Indexed	**sthx**	**r**S,**r**A,**r**B
Store Half Word with Update	**sthu**	**r**S,**d(r**A**)**
Store Half Word with Update Indexed	**sthux**	**r**S,**r**A,**r**B
Store Word	**stw**	**r**S,**d(r**A**)**
Store Word Indexed	**stwx**	**r**S,**r**A,**r**B
Store Word with Update	**stwu**	**r**S,**d(r**A**)**
Store Word with Update Indexed	**stwux**	**r**S,**r**A,**r**B

2.3.4.3.5 Integer Store Gathering

Gekko performs store gathering for write-through accesses to nonguarded space or to cache-inhibited stores to nonguarded space if the stores are 4 bytes and they are word-aligned. These stores are combined in the load/store unit (LSU) to form a double word and are sent out on the 60x bus as a single-beat operation. However, stores can be gathered only if the successive stores that meet the criteria are queued and pending. Store gathering takes place regardless of the address order of the stores. The store gathering feature is enabled by setting HID0[SGE]. Store gathering is done for both big- and little-endian modes.

Store gathering is not done for the following:

- Cacheable stores
- Stores to guarded cache-inhibited or write-through space
- Byte-reverse store
- **stwcx.** and **ecowx** accesses
- Floating-point stores
- Store operations attempted during a hardware table search

If store gathering is enabled and the stores do not fall under the above categories, an **eieio** or **sync** instruction must be used to prevent two stores from being gathered.

Note that the write gather pipe facility provides a separate mechanism for gathering operands before transferring them to memory. See Chapter 9 for a description of this facility.

2.3.4.3.6 Integer Load and Store with Byte-Reverse Instructions

Table 2-36 describes integer load and store with byte-reverse instructions. When used in a PowerPC system operating with the default big-endian byte order, these instructions have the effect of loading and storing data in little-endian order. Likewise, when used in a PowerPC system operating with little-endian byte order, these instructions have the effect of loading and storing data in big-endian order. For more information about big-endian and little-endian byte ordering, see “Byte Ordering" in Chapter 3, “Operand Conventions" in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Table 2-36. Integer Load and Store with Byte-Reverse Instructions

Name	Mnemonic	Syntax
Load Half Word Byte-Reverse Indexed	**lhbrx**	**r**D,**r**A,**r**B
Load Word Byte-Reverse Indexed	**lwbrx**	**r**D,**r**A,**r**B
Store Half Word Byte-Reverse Indexed	**sthbrx**	**r**S,**r**A,**r**B
Store Word Byte-Reverse Indexed	**stwbrx**	**r**S,**r**A,**r**B

2.3.4.3.7 Integer Load and Store Multiple Instructions

The load/store multiple instructions are used to move blocks of data to and from the GPRs. The load multiple and store multiple instructions may have operands that require memory accesses crossing a 4-Kbyte page boundary. As a result, these instructions may be interrupted by a DSI exception associated with the address translation of the second page.

Implementation Notes—The following describes the Gekko implementation of the load/store multiple instruction:

- For load/store string operations, the hardware does not combine register values to reduce the number of discrete accesses. However, if store gathering is enabled and the accesses fall under the criteria for store gathering the stores may be combined to enhance performance. At a minimum, additional cache access cycles are required.
- Gekko supports misaligned, single-register load and store accesses in little-endian mode without causing an alignment exception. However, execution of misaligned load/store multiple/string operations causes an alignment exception.

The PowerPC architecture defines the load multiple word (**lmw**) instruction with **r**A in the range of registers to be loaded as an invalid form.

Table 2-37. Integer Load and Store Multiple Instructions

Name	Mnemonic	Syntax
Load Multiple Word	**lmw**	**r**D,**d(r**A**)**
Store Multiple Word	**stmw**	**r**S,**d(r**A**)**

2.3.4.3.8 Integer Load and Store String Instructions

The integer load and store string instructions allow movement of data from memory to registers or from registers to memory without concern for alignment. These instructions can be used for a short move between arbitrary memory locations or to initiate a long move between misaligned memory fields. However, in some implementations, these instructions are likely to have greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Table 2-36 summarizes the integer load and store string instructions. In other PowerPC implementations operating with little-endian byte order, execution of a load or string instruction invokes the alignment error handler; see “Byte Ordering" in the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

Table 2-38. Integer Load and Store String Instructions

Name	Mnemonic	Syntax
Load String Word Immediate	**lswi**	**r**D,**r**A,NB
Load String Word Indexed	**lswx**	**r**D,**r**A,**r**B
Store String Word Immediate	**stswi**	**r**S,**r**A,NB
Store String Word Indexed	**stswx**	**r**S,**r**A,**r**B

Load string and store string instructions may involve operands that are not word-aligned.

As described in Section 4.5.6, "Alignment Exception (0x00600)" on Page 4-19, a misaligned string operation suffers a performance penalty compared to an aligned operation of the same type.

A non–word-aligned string operation that crosses a 4-Kbyte boundary, or a word-aligned string operation that crosses a 256-Mbyte boundary always causes an alignment exception. A non–word-aligned string operation that crosses a double-word boundary is also slower than a word-aligned string operation.

Implementation Note—The following describes the Gekko implementation of load/store string instructions:

- For load/store string operations, the hardware does not combine register values to reduce the number of discrete accesses. However, if store gathering is enabled and the accesses fall under the criteria for store gathering the stores may be combined to enhance performance. At a minimum, additional cache access cycles are required.
- Gekko supports misaligned, single-register load and store accesses in little-endian mode without causing an alignment exception. However, execution of misaligned load/store multiple/string operations cause an alignment exception.

2.3.4.3.9 Floating-Point Load and Store Address Generation

Floating-point load and store operations generate effective addresses using the register indirect with immediate index addressing mode and register indirect with index addressing mode. Floating-point loads and stores are not supported for direct-store accesses. The use of floating-point loads and stores for direct-store access results in an alignment exception.

Implementation Notes—Gekko treats exceptions as follows:

- The FPU can be run in two different modes—ignore exceptions mode (MSR[FE0] = MSR[FE1] = 0) and precise mode (any other settings for MSR[FE0,FE1]). For Gekko, ignore exceptions mode allows floating-point instructions to complete earlier and thus may provide better performance than precise mode.
- The floating-point load and store indexed instructions (**lfsx**, **lfsux**, **lfdx**, **lfdux**, **stfsx**, **stfsux**, **stfdx**, **stfdux**) are invalid when the Rc bit is one. In Gekko, executing one of these invalid instruction forms causes CR0 to be set to an undefined value.

2.3.4.3.10 Floating-Point Load Instructions

There are three forms of the floating-point load instruction—single-precision, double-precision and paired single (quantized) operand formats. The behavior of double- precision floating-point load instructions, and the behavior of single-precision floating- point load instructions when HID2[PSE] = 0 are described here. Paired single floating-point load instructions are illegal when HID2[PSE] = 0. The behavior of single-precision floating-point load instructions and paired single (quantized) load instructions when HID2[PSE] = 1 are described in Section 2.3.4.3.12, "Paired Single Load and Store Instructions" on Page 2-55.

Single-precision floating-point load instructions convert single-precision data to double-precision format before loading an operand into an FPR.

The PowerPC architecture defines a load with update instruction with **r**A = 0 as an invalid form.

Table 2-39 summarizes the single- and double-precision floating-point load instructions.

Table 2-39. Floating-Point Load Instructions

Name	Mnemonic	Syntax
Load Floating-Point Single	**lfs**	**fr**D,**d(r**A**)**
Load Floating-Point Single Indexed	**lfsx**	**fr**D,**r**A,**r**B
Load Floating-Point Single with Update	**lfsu**	**fr**D,**d(r**A**)**
Load Floating-Point Single with Update Indexed	**lfsux**	**fr**D,**r**A,**r**B
Load Floating-Point Double	**lfd**	**fr**D,**d(r**A**)**
Load Floating-Point Double Indexed	**lfdx**	**fr**D,**r**A,**r**B
Load Floating-Point Double with Update	**lfdu**	**fr**D,**d(r**A**)**
Load Floating-Point Double with Update Indexed	**lfdux**	**fr**D,**r**A,**r**B

2.3.4.3.11 Floating-Point Store Instructions

This section describes floating-point store instructions. There are four basic forms of the store instruction—single-precision, double-precision, paired single (quantized) and integer. The integer form is supported by the optional **stfiwx** instruction. The behavior of double- precision floating-point store instructions, and the behavior of single-precision floating- point store instructions when HID2[PSE] = 0 are described here. Paired single floating-point store instructions are illegal when HID2[PSE] = 0. The behavior of single-precision floating-point store instructions and paired single (quantized) store instructions when HID2[PSE] = 1 is described in Section 2.3.4.3.12, "Paired Single Load and Store Instructions" on Page 2-55. Single-precision floating-point store instructions convert double-precision data to single-precision format before storing the operands.

Table 2-40 summarizes the single- and double-precision floating-point store and **stfiwx** instructions. Some floating-point store instructions require conversions in the LSU.

Table 2-40. Floating-Point Store Instructions

Name	Mnemonic	Syntax
Store Floating-Point Single	**stfs**	**fr**S**,d(r**A**)**
Store Floating-Point Single Indexed	**stfsx**	**fr**S**,r** B
Store Floating-Point Single with Update	**stfsu**	**fr**S**,d(**rA**)**
Store Floating-Point Single with Update Indexed	**stfsux**	**fr**S**,r** B
Store Floating-Point Double	**stfd**	**fr**S**,d(r**A**)**
Store Floating-Point Double Indexed	**stfdx**	**fr**S**,r**B
Store Floating-Point Double with Update	**stfdu**	**fr**S**,d(**rA**)**
Store Floating-Point Double with Update Indexed	**stfdux**	**fr**S**,r** B
Store Floating-Point as Integer Word Indexed [1]	**stfiwx**	**fr**S**,r**B

Note: [1]The **stfiwx** instruction is optional to the PowerPC architecture.

Table 2-41 shows conversions the LSU makes when executing a Store Floating-Point Single instruction (when HID2[PSE] = 0).

Table 2-41. Store Floating-Point Single Behavior

FPR Precision	Data Type	Action
Single	Normalized	Store
Single	Denormalized	Store
Single	Zero, infinity, QNaN	Store
Single	SNaN	Store

Double	Normalized	If(exp ≤ 896)
		then Denormalize and Store
		else
		Store

Double	Denormalized	Store zero
Double	Zero, infinity, QNaN	Store
Double	SNaN	Store

NOTE: The FPRs are not initialized by $\overline{\text{HRESET}}$, and they must be initialized with some valid value after POR $\overline{\text{HRESET}}$ and before being stored.

Table 2-42 shows the conversions made when performing a Store Floating-Point Double instruction. Most entries in the table indicate that the floating-point value is simply stored. Only in a few cases are any other actions taken.

Table 2-42. Store Floating-Point Double Behavior

FPR Precision	Data Type	Action
Single	Normalized	Store
Single	Denormalized	Normalize and Store
Single	Zero, infinity, QNaN	Store
Single	SNaN	Store
Double	Normalized	Store
Double	Denormalized	Store
Double	Zero, infinity, QNaN	Store
Double	SNaN	Store

Architecturally, all single- and double-precision floating-point numbers are represented in double-precision format within Gekko. Execution of a store floating-point single (**stfs**, **stfsu**, **stfsx**, **stfsux**) instruction requires conversion from double- to single-precision format. If the exponent is not greater than 896, this conversion requires denormalization. Gekko supports this denormalization by shifting the mantissa one bit at a time. Anywhere from 1 to 23 clock cycles are required to complete the denormalization, depending upon the value to be stored.

Because of how floating-point numbers are implemented in Gekko, there is also a case when execution of a store floating-point double (**stfd**, **stfdu**, **stfdx**, **stfdux**) instruction can require internal shifting of the mantissa. This case occurs when the operand of a store floating-point double instruction is a denormalized single-precision value. The value could be the result of a load floating-point single instruction, a single-precision arithmetic instruction, or a floating round to single-precision instruction. In these cases, shifting the mantissa takes from 1 to 23 clock cycles, depending upon the value to be stored. These cycles are incurred during the store.

2.3.4.3.12 Paired Single Load and Store Instructions

In addition to the floating-point load and store instructions defined in the PowerPC architecture, Gekko includes eight additional load and store instructions that can implicitly convert their operands between single-precision floating-point and lower precision, quantized data types. For load instructions, this conversion is an inverse quantization, or dequantization, operation that converts signed or unsigned, 8 or 16 bit integers to 32 bit single-precision floating-point operands. This conversion takes place in the load/store unit as the data is being transfered to a floating-point register (FPR). For store instructions, the conversion is a quantization operation that converts single-precision floating-point numbers to operands having one of the quantized data types. This conversion takes place in the load/store unit as the data is transfered out of an FPR.

The load and store instructions for which data quantization applies are for ‘paired single’ operands, and so are valid only when HID2[PSE] = 1. These new load and store instructions cause an illegal instruction exception if execution is attempted when HID2[PSE] = 0. Furthermore, the nonindexed forms of these loads and stores (**psq_l**[**u**] and **psq_st**[**u**]) are illegal unless HID2[LSQE] = 1 as well. The quantization/dequantization hardware in the load/store unit assumes big-endian ordering of the data in memory. Use of these instructions in little-endian mode (MSR[LE] = 1) will give

undefined results. Whenever a pair of operands are converted, they are both converted in the same manner.

When operating in paired single mode (HID2[PSE] = 1), the behavior of single-precision floating-point load and store instructions is different from that described in the previous two sections. In this mode, a single-precision floating-point load instruction will load one single-precision operand into both the high and low order words of the operand pair in an FPR. A single-precision floating-point store instruction will store only the high order word of the operand pair in an FPR.

Table 2-43 summarizes the paired single load and store instructions.

Table 2-43. Paired Single Load and Store Instructions

Name	Mnemonic	Syntax
Paired Single Quantized Load [2]	**psq_l**	**fr**D,**d(r**A**),**W,**qr**I
Paired Single Quantized Load Indexed [1]	**psq_lx**	**fr**D,**r**A,**r**B,W,**qr**I
Paired Single Quantized Load with Update [2]	**psq_lu**	**fr**D,**d(r**A**),**W,**qr**I
Paired Single Quantized Load with Update Indexed [1]	**psq_lux**	**fr**D,**r**A,**r**B,W,**qr**I
Paired Single Quantized Store [2]	**psq_st**	**fr**S,**d(r**A**),**W,**qr**I
Paired Single Quantized Store Indexed [1]	**psq_stx**	**fr**S,**r**A,**r**B,W,**qr**I
Paired Single Quantized Store with Update [2]	**psq_stu**	**fr**S,**d(r**A**),**W,**qr**I
Paired Single Quantized Store with Update Indexed [1]	**psq_stux**	**fr**S,**r**A,**r**B,W,**qr**I

Note: [1]These instructions belong to the Gekko graphics extensions, and are legal only when HID2[PSE] = 1.
Note: [2]These instructions belong to the Gekko graphics extensions, and are legal only when HID2[PSE] = 1 and HID2[LSQE] = 1.

Two paired single load (**psq_l**, **psq_lu**) and two paired single store (**psq_st**, **psq_stu**) instructions use a variation of the D-form instruction format. Instead of having a 16 bit displacement field, 12 bits are used for displacement, and the remaining four are used to specify whether one or two operands are to be processed (the 1 bit W field) and which of the eight GQRs is to be used to specify the scale and type for the conversion (the 3 bit I field). The two remaining paired single load (**psq_lx**, **psq_lux**) and the two remaining paired single store (**psq_stx**, **psq_stux**) instructions use a variation of the X-form instruction format. Instead of having a 10 bit secondary opcode field, 6 bits are used for the secondary opcode, and the remaining four are used for the W field and the I field.

See Chapter 12, "Instruction Set" in this manual for more information on the instruction format.

The dequantization algorithm used to convert each integer of a pair to a single-precision floating-point operand is as follows:

1. read integer operand from L1 cache
2. convert data to sign and magnitude according to type specified in the selected GQR
3. convert magnitude to normalized mantissa and exponent
4. subtract scaling factor specified in the selected GQR from the exponent
5. load the converted value into the target FPR

For an integer value, I, in memory, the floating-point value F, loaded into the target FPR, is F = I * 2**(-S), where S is the twos compliment value in the LD_SCALE field of the selected GQR.

Table 2-44 shows how an integer value of 1 is converted to a single-precision floating-point value for various scaling factors.

Table 2-44. Conversion of integer value 1 to single-precision floating point

GQRx[LD_SCALE]	scaling factor (S)	floating-point value
100000	-32	4.29 E+9
100001	-31	2.15 E+9
...		
111110	-2	4.00 E+0
111111	-1	2.00 E+0
000000	0	1.00 E+0
000001	1	5.00 E-1
000010	2	2.50 E-1
...		
011110	30	9.31 E-10
011111	31	4.66 E-10

For a single-precision floating-point operand (type = 0), the value from the L1 cache is passed directly to the register without any conversion. This includes the case where the operand is a denorm.

The quantization algorithm used to convert each single-precision floating-point operand of a pair to an integer is as follows:

1. Move the single-precision floating-point operand from the FPR to the completion store queue.
2. Add the scaling factor specified in the selected GQR to the exponent
3. Shift mantissa and increment/decrement exponent until exponent is zero
4. Convert sign and magnitude to 2s complement representation, and
5. Round toward zero to get the type specified in the selected GQR
6. Adjust the resulting value on overflow
7. Store the converted value in the L1 cache.

The adjusted result value for overflow of unsigned integers is zero for negative values, 255 and 65535 for positive values, for 8 and 16 bit types, respectively. The adjusted result value for overflow of signed integers is -128 and -32768 for negative values, 127 and 32767 for positive values, for 8 and 16 bit types, respectively. The converted value produced when the input operand is +Inf or NaN is the same as the adjusted result value for overflow of positive values for the target data type. The converted value produced when the input operand is -Inf is the same as the adjusted result value for

overflow of negative values.

For a single-precision floating-point value, F, in an FPR, the integer value I, stored to memory, is I = ROUND(F * 2**(S)), where S is the twos compliment value in the ST_SCALE field of the selected GQR, and ROUND applies the rounding and clamping appropriate to the particular target integer format.

Table 2-45 shows how a floating-point value of 1.00 E+2 is converted to an integer value for various scaling factors.

Table 2-45. Conversion of Floating-point Value 1.00 E+2 to Integer

GQRx[LD_SCALE]	scaling factor (S)	u8 value	u16	s8	s16
100000	-32	0	0	0	0
100001	-31	0	0	0	0
...					
111110	-2	25	25	25	25
111111	-1	50	50	50	50
000000	0	100	100	100	100
000001	1	200	200	127	200
000010	2	255	400	127	400
...					
011110	30	255	65535	127	32767
011111	31	255	65525	127	32767

For a single-precision floating-point operand (type = 0), the value from the FPR is passed directly to the L1 cache without any conversion, except when this operand is a denorm. In the case of a denorm, the value 0.0 is stored in the L1 cache.

2.3.4.4 Branch and Flow Control Instructions

Some branch instructions can redirect instruction execution conditionally based on the value of bits in the CR. When the processor encounters one of these instructions, it scans the execution pipelines to determine whether an instruction in progress may affect the particular CR bit. If no interlock is found, the branch can be resolved immediately by checking the bit in the CR and taking the action defined for the branch instruction.

2.3.4.4.1 Branch Instruction Address Calculation

Branch instructions can alter the sequence of instruction execution. Instruction addresses are always assumed to be word aligned; the PowerPC processors ignore the two low-order bits of the generated branch target address.

Branch instructions compute the EA of the next instruction address using the following addressing modes:

- Branch relative
- Branch conditional to relative address
- Branch to absolute address
- Branch conditional to absolute address
- Branch conditional to link register
- Branch conditional to count register

Note that in Gekko, all branch instructions (**b**, **ba**, **bl**, **bla**, **bc**, **bca**, **bcl**, **bcla**, **bclr**, **bclrl**, **bcctr**, **bcctrl**) and condition register logical instructions (**crand**, **cror**, **crxor**, **crnand**, **crnor**, **crandc**, **creqv**, **crorc**, and **mcrf**) are executed by the BPU. Some of these instructions can redirect instruction execution conditionally based on the value of bits in the CR. Whenever the CR bits resolve, the branch direction is either marked as correct or mispredicted. Correcting a mispredicted branch requires that Gekko flush speculatively executed instructions and restore the machine state to immediately after the branch. This correction can be done immediately upon resolution of the condition registers bits.

2.3.4.4.2 Branch Instructions

Table 2-46 lists the branch instructions provided by the PowerPC processors. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions.

See Appendix F, “Simplified Mnemonics" in the *PowerPC Microprocessor Family: The Programming Environments* manual for a list of simplified mnemonic examples.

Table 2-46. Branch Instructions

Name	Mnemonic	Syntax
Branch	**b (ba bl bla)**	target_addr
Branch Conditional	**bc (bca bcl bcla)**	BO**,**BI,target_addr
Branch Conditional to Link Register	**bclr (bclrl)**	BO**,**BI
Branch Conditional to Count Register	**bcctr (bcctrl)**	BO**,**BI

2.3.4.4.3 Condition Register Logical Instructions

Condition register logical instructions and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions.

Table 2-47 shows these instructions.

Table 2-47. Condition Register Logical Instructions

Name	Mnemonic	Syntax
Condition Register AND	**crand**	**crb**D**,crb**A**,crb**B
Condition Register OR	**cror**	**crb**D**,crb**A**,crb**B

Table 2-47. Condition Register Logical Instructions (Continued)

Name	Mnemonic	Syntax
Condition Register XOR	**crxor**	**crb**D,**crb**A,**crb**B
Condition Register NAND	**crnand**	**crb**D,**crb**A,**crb**B
Condition Register NOR	**crnor**	**crb**D,**crb**A,**crb**B
Condition Register Equivalent	**creqv**	**crb**D,**crb**A, **crb**B
Condition Register AND with Complement	**crandc**	**crb**D,**crb**A, **crb**B
Condition Register OR with Complement	**crorc**	**crb**D,**crb**A, **crb**B
Move Condition Register Field	**mcrf**	**crf**D,**crf**S

NOTE: If the LR update option is enabled for any of these instructions, the PowerPC architecture defines these forms of the instructions as invalid.

2.3.4.4.4 Trap Instructions

The trap instructions shown in Table 2-48 are provided to test for a specified set of conditions. If any of the conditions tested by a trap instruction are met, the system trap type program exception is taken. For more information, see Section 4.5.7, "Program Exception (0x00700)" on Page 4-19. If the tested conditions are not met, instruction execution continues normally.

Table 2-48. Trap Instructions

Name	Mnemonic	Syntax
Trap Word Immediate	**twi**	TO,rA,SIMM
Trap Word	**tw**	TO,**r**A,**r**B

See Appendix F, “Simplified Mnemonics" in the *PowerPC Microprocessor Family: The Programming Environments* manual for a complete set of simplified mnemonics.

2.3.4.5 System Linkage Instruction—UISA

The System Call (sc) instruction permits a program to call on the system to perform a service; see Table 2-49. See also Section 2.3.6.1, "System Linkage Instructions—OEA" on Page 2-70 for additional information.

Table 2-49. System Linkage Instruction—UISA

Name	Mnemonic	Syntax
System Call	**sc**	—

Executing this instruction causes the system call exception handler to be evoked. For more information, see Section 4.5.10, "System Call Exception (0x00C00)" on Page 4-20.

2.3.4.6 Processor Control Instructions—UISA

Processor control instructions are used to read from and write to the condition register (CR), machine state register (MSR), and special-purpose registers (SPRs).

See Section 2.3.5.1, "Processor Control Instructions—VEA" on Page 2-65 for the **mftb** instruction and Section 2.3.6.2, "Processor Control Instructions—OEA" on Page 2-71 for information about the instructions used for reading from and writing to the MSR and SPRs.

2.3.4.6.1 Move to/from Condition Register Instructions

Table 2-50 summarizes the instructions for reading from or writing to the condition register.

Table 2-50. Move to/from Condition Register Instructions

Name	Mnemonic	Syntax
Move to Condition Register Fields	**mtcrf**	CRM,**r**S
Move to Condition Register from XER	**mcrxr**	**crf**D
Move from Condition Register	**mfcr**	**r**D

Implementation Note—The PowerPC architecture indicates that in some implementations the Move to Condition Register Fields (**mtcrf**) instruction may perform more slowly when only a portion of the fields are updated as opposed to all of the fields. The condition register access latency for Gekko is the same in both cases.

2.3.4.6.2 Move to/from Special-Purpose Register Instructions (UISA)

Table 2-51 lists the **mtspr** and **mfspr** instructions.

Table 2-51. Move to/from Special-Purpose Register Instructions (UISA)

Name	Mnemonic	Syntax
Move to Special-Purpose Register	**mtspr**	SPR,**r**S
Move from Special-Purpose Register	**mfspr**	**r**D,SPR

Table 2-52 lists the SPR numbers for both user- and supervisor-level accesses.

Table 2-52. PowerPC Encodings

Register Name	SPR[1]			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
CTR	9	00000	01001	User (UISA)	Both
DABR	1013	11111	10101	Supervisor (OEA)	Both
DAR	19	00000	10011	Supervisor (OEA)	Both
DBAT0L	537	10000	11001	Supervisor (OEA)	Both
DBAT0U	536	10000	11000	Supervisor (OEA)	Both
DBAT1L	539	10000	11011	Supervisor (OEA)	Both
DBAT1U	538	10000	11010	Supervisor (OEA)	Both

Table 2-52. PowerPC Encodings (Continued)

Register Name	SPR[1]			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
DBAT2L	541	10000	11101	Supervisor (OEA)	Both
DBAT2U	540	10000	11100	Supervisor (OEA)	Both
DBAT3L	543	10000	11111	Supervisor (OEA)	Both
DBAT3U	542	10000	11110	Supervisor (OEA)	Both
DEC	22	00000	10110	Supervisor (OEA)	Both
DSISR	18	00000	10010	Supervisor (OEA)	Both
EAR	282	01000	11010	Supervisor (OEA)	Both
IBAT0L	529	10000	10001	Supervisor (OEA)	Both
IBAT0U	528	10000	10000	Supervisor (OEA)	Both
IBAT1L	531	10000	10011	Supervisor (OEA)	Both
IBAT1U	530	10000	10010	Supervisor (OEA)	Both
IBAT2L	533	10000	10101	Supervisor (OEA)	Both
IBAT2U	532	10000	10100	Supervisor (OEA)	Both
IBAT3L	535	10000	10111	Supervisor (OEA)	Both
IBAT3U	534	10000	10110	Supervisor (OEA)	Both
LR	8	00000	01000	User (UISA)	Both
PVR	287	01000	11111	Supervisor (OEA)	**mfspr**
SDR1	25	00000	11001	Supervisor (OEA)	Both
SPRG0	272	01000	10000	Supervisor (OEA)	Both
SPRG1	273	01000	10001	Supervisor (OEA)	Both
SPRG2	274	01000	10010	Supervisor (OEA)	Both
SPRG3	275	01000	10011	Supervisor (OEA)	Both
SRR0	26	00000	11010	Supervisor (OEA)	Both
SRR1	27	00000	11011	Supervisor (OEA)	Both
TBL [2]	268	01000	01100	User (VEA)	**mfspr**
	284	01000	11100	Supervisor (OEA)	**mtspr**
TBU [2]	269	01000	01101	User (VEA)	**mfspr**
	285	01000	11101	Supervisor (OEA)	**mtspr**
XER	1	00000	00001	User (UISA)	Both

Notes:

[1] The order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding. For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order five bits appearing in bits 16–20 of the instruction and the low-order five bits in bits 11–15.

[2] The TB registers are referred to as TBRs rather than SPRs and can be written to using the **mtspr** instruction in supervisor mode and the TBR numbers here. The TB registers can be read in user mode using either the **mftb** or **mfspr** instruction and specifying TBR 268 for TBL and SPR 269 for TBU.

Encodings for the Gekko-specific SPRs are listed in Table 2-53.

Table 2-53. SPR Encodings for Gekko-Defined Registers (mfspr)

Register Name	SPR[1]			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
DABR	1013	11111	10101	User	Both
DMAL [2]	923	11100	11011	Supervisor	Both
DMAU [2]	922	11100	11010	Supervisor	Both
GQR0 [2]	912	11100	10000	Supervisor	Both
GQR1 [2]	913	11100	10001	Supervisor	Both
GQR2 [2]	914	11100	10010	Supervisor	Both
GQR3 [2]	915	11100	10011	Supervisor	Both
GQR4 [2]	916	11100	10100	Supervisor	Both
GQR5 [2]	917	11100	10101	Supervisor	Both
GQR6 [2]	918	11100	10110	Supervisor	Both
GQR7 [2]	919	11100	10111	Supervisor	Both
HID0	1008	11111	10000	Supervisor	Both
HID1	1009	11111	10001	Supervisor	Both
HID2 [2]	920	11100	11000	Supervisor	Both
IABR	1010	11111	10010	Supervisor	Both
ICTC	1019	11111	11011	Supervisor	Both
L2CR	1017	11111	11001	Supervisor	Both
MMCR0	952	11101	11000	Supervisor	Both
MMCR1	956	11101	11100	Supervisor	Both
PMC1	953	11101	11001	Supervisor	Both
PMC2	954	11101	11010	Supervisor	Both
PMC3	957	11101	11101	Supervisor	Both

Table 2-53. SPR Encodings for Gekko-Defined Registers (mfspr) (Continued)

Register Name	SPR[1]			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
PMC4	958	11101	11110	Supervisor	Both
SIA	955	11101	11011	Supervisor	Both
THRM1	1020	11111	11100	Supervisor	Both
THRM2	1021	11111	11101	Supervisor	Both
THRM3	1022	11111	11110	Supervisor	Both
UMMCR0	936	11101	01000	User	**mfspr**
UMMCR1	940	11101	01100	User	**mfspr**
UPMC1	937	11101	01001	User	**mfspr**
UPMC2	938	11101	01010	User	**mfspr**
UPMC3	941	11101	01101	User	**mfspr**
UPMC4	942	11101	01110	User	**mfspr**
USIA	939	11101	01011	User	**mfspr**
WPAR [2]	921	11100	11001	Supervisor	Both

Note:

[1]Note that the order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding.

For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16–20 of the instruction and the low-order 5 bits in bits 11–15.

[2]This register is part of the Gekko graphics extensions.

2.3.4.7 Memory Synchronization Instructions—UISA

Memory synchronization instructions control the order in which memory operations are completed with respect to asynchronous events, and the order in which memory operations are seen by other processors or memory access mechanisms. See Chapter 3, "Gekko Instruction and Data Cache Operation" in this manual for additional information about these instructions and about related

aspects of memory synchronization. See Table 2-54 for a summary.

Table 2-54. Memory Synchronization Instructions—UISA

Name	Mnemonic	Syntax	Implementation Notes
Load Word and Reserve Indexed	**lwarx**	**r**D,**r**A,**r**B	Programmers can use **lwarx** with **stwcx.** to emulate common semaphore operations such as test and set, compare and swap, exchange memory, and fetch and add. Both instructions must use the same EA. Reservation granularity is implementation-dependent. Gekko makes reservations on behalf of aligned 32-byte sections of the memory address space. If the W bit is set, executing **lwarx** and **stwcx.** to a page marked write-through does not cause a DSI exception, but DSI exceptions can result for other reasons. If the location is not word-aligned, an alignment exception occurs. The **stwcx.** instruction is the only load/store instruction with a valid form if Rc is set. If Rc is zero, executing **stwcx.** sets CR0 to an undefined value. In general, **stwcx.** always causes a transaction on the external bus and thus operates with slightly worse performance characteristics than normal store operations.
Store Word Conditional Indexed	**stwcx.**	**r**S,**r**A,**r**B	
Synchronize	**sync**	—	Because it delays subsequent instructions until all previous instructions complete to where they cannot cause an exception, **sync** is a barrier against store gathering when HID2[LCE] = 0 and HID2[WPE] = 0. See Chapter 9, "L2 Cache, Locked D-Cache, DMA and Write Gather Pipe" in this manual for a description of the modified **sync** behavior when HID2[LCE] = 1 or HID2[WPE] = 1. Additionally, all load/store cache/bus activities initiated by prior instructions are completed. Touch load operations (**dcbt**, **dcbtst**) must complete address translation, but need not complete on the bus. If HID0[ABE] = 1, **sync** completes after a successful broadcast. The latency of **sync** depends on the processor state when it is dispatched and on various system-level situations. Therefore, frequent use of **sync** may degrade performance.

System designs with an L2 cache should take special care to recognize the hardware signaling caused by a SYNC bus operation and perform the appropriate actions to guarantee that memory references that may be queued internally to the L2 cache have been performed globally.

See 2.3.5.2, “Memory Synchronization Instructions—VEA" for details about additional memory synchronization (**eieio** and **isync**) instructions.

In the PowerPC architecture, the Rc bit must be zero for most load and store instructions. If Rc is set, the instruction form is invalid for **sync** and **lwarx** instructions. If Gekko encounters one of these invalid instruction forms, it sets CR0 to an undefined value.

2.3.5 PowerPC VEA Instructions

The PowerPC virtual environment architecture (VEA) describes the semantics of the memory model that can be assumed by software processes, and includes descriptions of the cache model, cache control instructions, address aliasing, and other related issues. Implementations that conform to the VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

This section describes additional instructions that are provided by the VEA.

2.3.5.1 Processor Control Instructions—VEA

In addition to the move to condition register instructions (specified by the UISA), the VEA defines the **mftb** instruction (user-level instruction) for reading the contents of the time base register; see Chapter 3, "Gekko Instruction and Data Cache Operation" in this manual for more information.

Table 2-55 shows the **mftb** instruction.

Table 2-55. Move from Time Base Instruction

Name	Mnemonic	Syntax
Move from Time Base	**mftb**	**r**D, TBR

Simplified mnemonics are provided for the **mftb** instruction so it can be coded with the TBR name as part of the mnemonic rather than requiring it to be coded as an operand. See Appendix F, “Simplified Mnemonics" in the *PowerPC Microprocessor Family: The Programming Environments* manual for simplified mnemonic examples and for simplified mnemonics for Move from Time Base (**mftb**) and Move from Time Base Upper (**mftbu**), which are variants of the **mftb** instruction rather than of **mfspr**. The **mftb** instruction serves as both a basic and simplified mnemonic. Assemblers recognize an **mftb** mnemonic with two operands as the basic form, and an **mftb** mnemonic with one operand as the simplified form. Note that Gekko ignores the extended opcode differences between **mftb** and **mfspr** by ignoring bit 25 and treating both instructions identically.

Implementation Notes—The following information is useful with respect to using the time base implementation in Gekko:

- Gekko allows user-mode read access to the time base counter through the use of the Move from Time Base (**mftb**) and the Move from Time Base Upper (**mftbu**) instructions. As a 32-bit PowerPC implementation, Gekko can access TBU and TBL only separately, whereas 64-bit implementations can access the entire TB register at once.
- The time base counter is clocked at a frequency that is one-fourth that of the bus clock.

2.3.5.2 Memory Synchronization Instructions—VEA

Memory synchronization instructions control the order in which memory operations are completed with respect to asynchronous events, and the order in which memory operations are seen by other processors or memory access mechanisms. See Chapter 3, "Gekko Instruction and Data Cache Operation" in this manual for more information about these instructions and about related aspects of memory synchronization.

In addition to the **sync** instruction (specified by UISA), the VEA defines the Enforce In-Order Execution of I/O (**eieio**) and Instruction Synchronize (**isync**) instructions. The number of cycles required to complete an **eieio** instruction depends on system parameters and on the processor's state when the instruction is issued. As a result, frequent use of this instruction may degrade performance slightly.

Table 2-56 describes the memory synchronization instructions defined by the VEA.

Table 2-56. Memory Synchronization Instructions—VEA

Name	Mnemonic	Syntax	Implementation Notes
Enforce In-Order Execution of I/O	**eieio**	—	The **eieio** instruction is dispatched to the LSU and executes after all previous cache-inhibited or write-through accesses are performed; all subsequent instructions that generate such accesses execute after **eieio**. If HID0[ABE] = 1 an EIEIO operation is broadcast on the external bus to enforce ordering in the external memory system. The **eieio** operation bypasses the L2 cache and is forwarded to the bus unit. If HID0[ABE] = 0, the operation is not broadcast. Because Gekko does not reorder noncacheable accesses, **eieio** is not needed to force ordering. However, if store gathering is enabled and an **eieio** is detected in a store queue, stores are not gathered. If HID0[ABE] = 1, broadcasting **eieio** prevents external devices, such as a bus bridge chip, from gathering stores. The behavior of **eieio** is modified when either HID2[LCE] = 1 or HID2[WPE] = 1. See Chapter 9, "L2 Cache, Locked D-Cache, DMA and Write Gather Pipe" in this manual for a description of this modified behavior.
Instruction Synchronize	**isync**	—	The **isync** instruction is refetch serializing; that is, it causes Gekko to purge its instruction queue and wait for all prior instructions to complete before refetching the next instruction, which is not executed until all previous instructions complete to the point where they cannot cause an exception. The **isync** instruction does not wait for all pending stores in the store queue to complete. Any instruction after an **isync** sees all effects of prior instructions.

2.3.5.3 Memory Control Instructions—VEA

Memory control instructions can be classified as follows:

- Cache management instructions (user-level and supervisor-level)
- Segment register manipulation instructions (OEA)
- Translation lookaside buffer management instructions (OEA)

This section describes the user-level cache management instructions defined by the VEA. See Section 2.3.6.3, "Memory Control Instructions—OEA" on Page 2-71 for information about supervisor-level cache, segment register manipulation, and translation lookaside buffer management instructions.

2.3.5.3.1 User-Level Cache Instructions—VEA

The instructions summarized in this section help user-level programs manage on-chip caches if they are implemented. See Chapter 3, "Gekko Instruction and Data Cache Operation" in this manual for more information about cache topics. The following sections describe how these operations are treated with respect to Gekko's cache.

As with other memory-related instructions, the effects of cache management instructions on memory are weakly-ordered. If the programmer must ensure that cache or other instructions have been performed with respect to all other processors and system mechanisms, a **sync** instruction must be placed after those instructions.

Note that Gekko interprets cache control instructions (**icbi**, **dcbi**, **dcbf**, **dcbz**, and **dcbst**) as if they pertain only to the local L1 and L2 cache. A **dcbz** (with M set) is always broadcast on the 60x bus. The **dcbi**, **dcbf**, and **dcbst** operations are broadcast if HID0[ABE] is set.

Gekko never broadcasts an **icbi**. Of the broadcast cache operations, Gekko snoops only **dcbz**, regardless of the HID0[ABE] setting. Any bus activity caused by other cache instructions results

directly from performing the operation on the Gekko cache. All cache control instructions to T = 1 space are no-ops. For information on how cache control instructions affect the L2, see Chapter 9, "L2 Cache, Locked D-Cache, DMA and Write Gather Pipe" in this manual.

Table 2-57 summarizes the cache instructions defined by the VEA. Note that these instructions are accessible to user-level programs.

Table 2-57. User-Level Cache Instructions

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Touch [1]	**dcbt**	**r**A,**r**B	The VEA defines this instruction to allow for potential system performance enhancements through the use of software-initiated prefetch hints. Implementations are not required to take any action based on execution of this instruction, but they may prefetch the cache block corresponding to the EA into their cache. When **dcbt** executes, Gekko checks for protection violations (as for a load instruction). This instruction is treated as a no-op for the following cases: • A valid translation is not found either in BAT or TLB • The access causes a protection violation. • The page is mapped cache-inhibited, G = 1 (guarded), or T = 1. • The cache is locked or disabled • HID0[NOOPTI] = 1 Otherwise, if no data is in the cache location, Gekko requests a cache line fill (with intent to modify). Data brought into the cache is validated as if it were a load instruction. The memory reference of a **dcbt** sets the reference bit. The behavior of **dcbt** is modified when either HID2[LCE] = 1 or HID2[WPE] = 1. See Chapter 9, "L2 Cache, Locked D-Cache, DMA and Write Gather Pipe" in this manual for a description of this modified behavior.
Data Cache Block Touch for Store [1]	**dcbtst**	**r**A,**r**B	This instruction behaves like **dcbt**.
Data Cache Block Set to Zero	**dcbz**	**r**A,**r**B	The EA is computed, translated, and checked for protection violations. For cache hits, four beats of zeros are written to the cache block and the tag is marked M. For cache misses with the replacement block marked E, the zero line fill is performed and the cache block is marked M. However, if the replacement block is marked M, the contents are written back to memory first. The instruction executes regardless of whether the cache is locked; if the cache is disabled, an alignment exception occurs. If M = 1 (coherency enforced), the address is broadcast to the bus before the zero line fill. The exception priorities (from highest to lowest) are as follows: 1 Cache disabled—Alignment exception 2 Page marked write-through or cache Inhibited—Alignment exception 3 BAT protection violation—DSI exception 4 TLB protection violation—DSI exception **dcbz** is the only cache instruction that broadcasts even if HID0[ABE] = 0. The behavior of **dcbz** is modified when either HID2[LCE] = 1 or HID2[WPE] = 1. SeeChapter 9 for a description of this modified behavior.
Data Cache Block Set to Zero Locked	**dcbz_l**	**r**A,**r**B	This instruction is illegal when HID2[LCE] = 0. See Chapter 9, "L2 Cache, Locked D-Cache, DMA and Write Gather Pipe" in this manualfor a description of this instruction when HID2[LCE] = 1.

Table 2-57. User-Level Cache Instructions (Continued)

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Store	**dcbst**	**r**A,**r**B	The EA is computed, translated, and checked for protection violations. • For cache hits with the tag marked E, no further action is taken. • For cache hits with the tag marked M, the cache block is written back to memory and marked E. A **dcbst** is not broadcast unless HID0[ABE] = 1 regardless of WIMG settings. The instruction acts like a load with respect to address translation and memory protection. It executes regardless of whether the cache is disabled or locked. The exception priorities (from highest to lowest) for **dcbst** are as follows: 1 BAT protection violation—DSI exception 2 TLB protection violation—DSI exception The behavior of **dcbst** is modified when either HID2[LCE] = 1 or HID2[WPE] = 1. See Chapter 9 for a description of this modified behavior.
Data Cache Block Flush	**dcbf**	**r**A,**r**B	The EA is computed, translated, and checked for protection violations. • For cache hits with the tag marked M, the cache block is written back to memory and the cache entry is invalidated. • For cache hits with the tag marked E, the entry is invalidated. • For cache misses, no further action is taken. A **dcbf** is not broadcast unless HID0[ABE] = 1 regardless of WIMG settings. The instruction acts like a load with respect to address translation and memory protection. It executes regardless of whether the cache is disabled or locked. The exception priorities (from highest to lowest) for **dcbf** are as follows: 1 BAT protection violation—DSI exception 2 TLB protection violation—DSI exception The behavior of **dcbf** is modified when either HID2[LCE] = 1 or HID2[WPE] = 1. See Chapter 9 for a description of this modified behavior.
Instruction Cache Block Invalidate	**icbi**	**r**A,**r**B	This instruction performs a virtual lookup into the instruction cache (index only). The address is not translated, so it cannot cause an exception. All ways of a selected set are invalidated regardless of whether the cache is disabled or locked. Gekko never broadcasts **icbi** onto the 60x bus.

Note:

[1] A program that uses **dcbt** and **dcbtst** instructions improperly performs less efficiently. To improve performance, HID0[NOOPTI] may be set, which causes **dcbt** and **dcbtst** to be no-oped at the cache. They do not cause bus activity and cause only a 1-clock execution latency. The default state of this bit is zero which enables the use of these instructions.

2.3.5.4 Optional External Control Instructions

The PowerPC architecture defines an optional external control feature that, if implemented, is supported by the two external control instructions, **eciwx** and **ecowx**. These instructions allow a user-level program to communicate with a special-purpose device. These instructions are provided

and are summarized in Table 2-58.

Table 2-58. External Control Instructions

Name	Mnemonic	Syntax	Implementation Notes
External Control In Word Indexed	**eciwx**	**r**D,**r**A,**r**B	A transfer size of 4 bytes is implied; the $\overline{\text{TBST}}$ and TSIZ[0–2] signals are redefined to specify the Resource ID (RID), copied from bits EAR[28–31]. For these operations, $\overline{\text{TBST}}$ carries the EAR[28] data. Misaligned operands for these instructions cause an alignment exception. Addressing a location where SR[T] = 1 causes a DSI exception. If MSR[DR] = 0 a programming error occurs and the physical address on the bus is undefined. **Note**: These instructions are optional to the PowerPC architecture.
External Control Out Word Indexed	**ecowx**	**r**S,**r**A,**r**B	

The **eciwx**/**ecowx** instructions let a system designer map special devices in an alternative way. The MMU translation of the EA is not used to select the special device, as it is used in most instructions such as loads and stores. Rather, it is used as an address operand that is passed to the device over the address bus. Four other signals (the burst and size signals on the 60x bus) are used to select the device; these four signals output the 4-bit resource ID (RID) field located in the EAR. The **eciwx** instruction also loads a word from the data bus that is output by the special device. For more information about the relationship between these instructions and the system interface, refer to Chapter 7, "Signal Descriptions" in this manual.

2.3.6 PowerPC OEA Instructions

The PowerPC operating environment architecture (OEA) includes the structure of the memory management model, supervisor-level registers, and the exception model. Implementations that conform to the OEA also adhere to the UISA and the VEA. This section describes the instructions provided by the OEA.

2.3.6.1 System Linkage Instructions—OEA

This section describes the system linkage instructions (see Table 2-59). The user-level **sc** instruction lets a user program call on the system to perform a service and causes the processor to take a system call exception. The supervisor-level **rfi** instruction is used for returning from an exception handler.

Table 2-59. System Linkage Instructions—OEA

Name	Mnemonic	Syntax	Implementation Notes
System Call	**sc**	—	The **sc** instruction is context-synchronizing.
Return from Interrupt	**rfi**	—	The **rfi** instruction is context-synchronizing. For Gekko, this means the **rfi** instruction works its way to the final stage of the execution pipeline, updates architected registers, and redirects the instruction flow.

2.3.6.2 Processor Control Instructions—OEA

This section describes the processor control instructions used to access the MSR and the SPRs. Table 2-60 lists instructions for accessing the MSR.

Table 2-60. Move to/from Machine State Register Instructions

Name	Mnemonic	Syntax
Move to Machine State Register	**mtmsr**	**r**S
Move from Machine State Register	**mfmsr**	**r**D

The OEA defines encodings of **mtspr** and **mfspr** to provide access to supervisor-level registers. The instructions are listed in Table 2-61.

Table 2-61. Move to/from Special-Purpose Register Instructions (OEA)

Name	Mnemonic	Syntax
Move to Special-Purpose Register	**mtspr**	SPR,**r**S
Move from Special-Purpose Register	**mfspr**	**r**D,SPR

Encodings for the architecture-defined SPRs are listed in Table 2-53 on Page 2-63. Encodings for Gekko-specific, supervisor-level SPRs are listed in Table 2-54 on Page 2-65. Simplified mnemonics are provided for **mtspr** and **mfspr** in Appendix F, “Simplified Mnemonics" in the *PowerPC Microprocessor Family: The Programming Environments* manual.

For a discussion of context synchronization requirements when altering certain SPRs, refer to Appendix E, “Synchronization Programming Examples" in the *PowerPC Microprocessor Family: The Programming Environments* manual.

2.3.6.3 Memory Control Instructions—OEA

Memory control instructions include the following:

- Cache management instructions (supervisor-level and user-level)
- Segment register manipulation instructions
- Translation lookaside buffer management instructions

This section describes supervisor-level memory control instructions. Section 2.3.5.3, "Memory Control Instructions—VEA" on Page 2-67 describes user-level memory control instructions.

2.3.6.3.1 Supervisor-Level Cache Management Instruction—(OEA)

Table 2-62 lists the only supervisor-level cache management instruction.

Table 2-62. Supervisor-Level Cache Management Instruction

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Invalidate	**dcbi**	**r**A,**r**B	The EA is computed, translated, and checked for protection violations. For cache hits, the cache block is marked I regardless of whether it was marked E or M. A **dcbi** is not broadcast unless HID0[ABE] = 1, regardless of WIMG settings. The instruction acts like a store with respect to address translation and memory protection. It executes regardless of whether the cache is disabled or locked. The exception priorities (from highest to lowest) for **dcbi** are as follows: 1 BAT protection violation—DSI exception 2 TLB protection violation—DSI exception The behavior of **dcbi** is modified when either HID2[LCE] = 1 or HID2[WPE] = 1. See Chapter 9 for a description of this modified behavior.

See Section 2.3.5.3.1, "User-Level Cache Instructions—VEA" on Page 2-67 for cache instructions that provide user-level programs the ability to manage the on-chip caches. If the effective address references a direct-store segment, the instruction is treated as a no-op.

2.3.6.3.2 Segment Register Manipulation Instructions (OEA)

The instructions listed in Table 2-63 provide access to the segment registers for 32-bit implementations. These instructions operate completely independently of the MSR[IR] and MSR[DR] bit settings. Refer to “Synchronization Requirements for Special Registers and for Lookaside Buffers" in Chapter 2, “PowerPC Register Set" of the *PowerPC Microprocessor Family: The Programming Environments* manual for serialization requirements and other recommended precautions to observe when manipulating the segment registers.

Table 2-63. Segment Register Manipulation Instructions

Name	Mnemonic	Syntax	Implementation Notes
Move to Segment Register	**mtsr**	SR,**r**S	—
Move to Segment Register Indirect	**mtsrin**	**r**S,**r**B	—
Move from Segment Register	**mfsr**	**r**D,SR	The shadow SRs in the instruction MMU can be read by setting HID0[RISEG] before executing **mfsr**.
Move from Segment Register Indirect	**mfsrin**	**r**D,**r**B	—

2.3.6.3.3 Translation Lookaside Buffer Management Instructions—(OEA)

The address translation mechanism is defined in terms of the segment descriptors and page table entries (PTEs) PowerPC processors use to locate the logical-to-physical address mapping for a particular access. These segment descriptors and PTEs reside in segment registers and page tables in memory, respectively.

See Chapter 7, "Signal Descriptions" in this manual for more information about TLB operations.

Table 2-64 summarizes the operation of the TLB instructions in Gekko.

Table 2-64. Translation Lookaside Buffer Management Instruction

Name	Mnemonic	Syntax	Implementation Notes
TLB Invalidate Entry	**tlbie**	**r**B	Invalidates both ways in both instruction and data TLB entries at the index provided by EA[14–19]. It executes regardless of the MSR[DR] and MSR[IR] settings.To invalidate all entries in both TLBs, the programmer should issue 64 **tlbie** instructions that each successively increment this field.
TLB Synchronize	**tlbsync**	—	On Gekko, the only function **tlbsync** serves is to wait for the $\overline{\text{TLBISYNC}}$ signal to go inactive.

Implementation Note—The **tlbia** instruction is optional for an implementation if its effects can be achieved through some other mechanism. Therefore, it is not implemented on Gekko. As described above, **tlbie** can be used to invalidate a particular index of the TLB based on EA[14–19]—a sequence of 64 **tlbie** instructions followed by a **tlbsync** instruction invalidates all the TLB structures (for EA[14–19] = 0, 1, 2,..., 63). Attempting to execute **tlbia** causes an illegal instruction program exception.

The presence and exact semantics of the TLB management instructions are implementation-dependent. To minimize compatibility problems, system software should incorporate uses of these instructions into subroutines.

2.3.7 Recommended Simplified Mnemonics

To simplify assembly language coding, a set of alternative mnemonics is provided for some frequently used operations (such as no-op, load immediate, load address, move register, and complement register). Programs written to be portable across the various assemblers for the PowerPC architecture should not assume the existence of mnemonics not described in this document.

For a complete list of simplified mnemonics, see Appendix F, “Simplified Mnemonics" in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Chapter 3 Gekko Instruction and Data Cache Operation

Gekko microprocessor contains separate 32-Kbyte, eight-way set associative instruction and data caches to allow the execution units and registers rapid access to instructions and data. This chapter describes the organization of the on-chip instruction and data caches, the MEI cache coherency protocol, cache control instructions, various cache operations, and the interaction between the caches, the load/store unit (LSU), the instruction unit, and the bus interface unit (BIU).

At power-on, Gekko sets HID2[LCE] = 0 and the corresponding L1 data cache's operation is described in this chapter. When a **mtspr** instruction sets HID2[LCE] = 1, the L1 data cache is partitioned as a 16 Kbyte normal cache and a 16 Kbyte locked cache.

The operation is described in Chapter 9, "L2 Cache, Locked D-Cache, DMA and Write Gather Pipe" of this manaul. Also, in Gekko, locked cache and bus snoop are incompatible. HID2[LCE] shall be kept at 0 for systems which generate snoop transactions.

Note that in this chapter, the term 'multiprocessor' is used in the context of maintaining cache coherency. These multiprocessor devices could be actual processors or other devices that can access system memory, maintain their own caches, and function as bus masters requiring cache coherency.

The Gekko cache implementation has the following characteristics:

- There are two separate 32-Kbyte instruction and data caches (Harvard architecture).
- Both instruction and data caches are eight-way set associative.
- The caches implement a pseudo least-recently-used (PLRU) replacement algorithm within each set.
- The cache directories are physically addressed. The physical (real) address tag is stored in the cache directory.
- Both the instruction and data caches have 32-byte cache blocks. A cache block is the block of memory that a coherency state describes, also referred to as a cache line.
- Two coherency state bits for each data cache block allow encoding for three states:
  - — Modified (Exclusive) (M)
  - — Exclusive (Unmodified) (E)
  - — Invalid (I)
- A single coherency state bit for each instruction cache block allows encoding for two possible states:
  - — Invalid (INV)
  - — Valid (VAL)
- Each cache can be invalidated or locked by setting the appropriate bits in the hardware implementation-dependent register 0 (HID0), a special-purpose register (SPR) specific to Gekko.

Gekko supports a fully-coherent 4-Gbyte physical memory address space. Bus snooping is used to drive the MEI three-state cache coherency protocol that ensures the coherency of global memory with respect to the processor's data cache. The MEI protocol is described in 3.3.2."

On a cache miss, Gekko's cache blocks are filled in four beats of 64 bits each. The burst fill is performed as a critical-double-word-first operation; the critical double word is simultaneously

written to the cache and forwarded to the requesting unit, thus minimizing stalls due to cache fill latency.

The instruction and data caches are integrated into Gekko as shown in Figure 3-1.

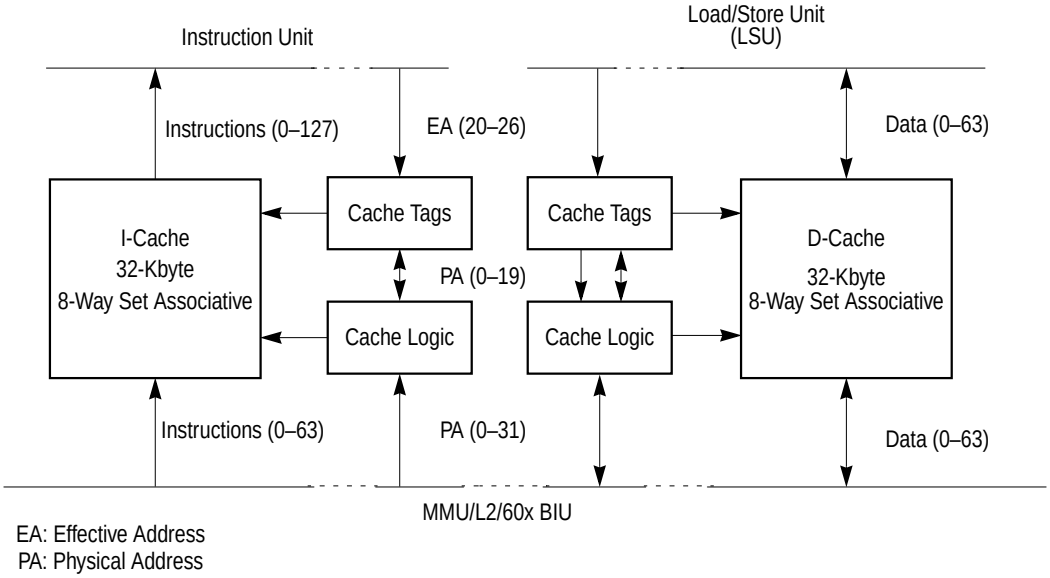


Figure 3-1. Cache Integration

Both caches are tightly coupled into Gekko's bus interface unit to allow efficient access to the system memory controller and other bus masters. The bus interface unit receives requests for bus operations from the instruction and data caches, and executes the operations per the 60x bus protocol. The BIU provides address queues, prioritizing logic, and bus control logic. The BIU captures snoop addresses for data cache, address queue, and memory reservation (**lwarx** and **stwcx.** instruction) operations.

The data cache provides buffers for load and store bus operations. All the data for the corresponding address queues (load and store data queues) is located in the data cache. The data queues are considered temporary storage for the cache and not part of the BIU. The data cache also provides storage for the cache tags required for memory coherency and performs the cache block replacement PLRU function.

The data cache supplies data to the GPRs and FPRs by means of the load/store unit. Gekko's LSU is directly coupled to the data cache to allow efficient movement of data to and from the general-purpose and floating-point registers. The load/store unit provides all logic required to calculate effective addresses, handles data alignment to and from the data cache, and provides sequencing for load and store string and multiple operations. Write operations to the data cache can be performed on a byte, half-word, word, or double-word basis.

The instruction cache provides a 128-bit interface to the instruction unit, so four instructions can be made available to the instruction unit in a single clock cycle. The instruction unit accesses the instruction cache frequently in order to sustain the high throughput provided by the six-entry instruction queue.

3.1 Data Cache Organization

The data cache is organized as 128 sets of eight ways as shown in Figure 3-2. Each way consists of 32 bytes, two state bits, and an address tag. Note that in the PowerPC architecture, the term ‘cache block,’ or simply ‘block,’ when used in the context of cache implementations, refers to the unit of memory at which coherency is maintained. For Gekko, this is the eight-word (32 byte) cache line. This value may be different for other PowerPC implementations.

Each cache block contains eight contiguous words from memory that are loaded from an eight-word boundary (that is, bits A[27–31] of the logical (effective) addresses are zero); as a result, cache blocks are aligned with page boundaries. Note that address bits A[20–26] provide the index to select a cache set. Bits A[27–31] select a byte within a block. The two state bits implement a three-state MEI (modified/exclusive/invalid) protocol, a coherent subset of the standard four-state MESI (modified/exclusive/shared/invalid) protocol. The MEI protocol is described in 3.3.2.” The tags consist of bits PA[0–19]. Address translation occurs in parallel with set selection (from A[20–26]), and the higher-order address bits (the tag bits in the cache) are physical.

Gekko’s on-chip data cache tags are single-ported, and load or store operations must be arbitrated with snoop accesses to the data cache tags. Load or store operations can be performed to the cache on the clock cycle immediately following a snoop access if the snoop misses; snoop hits may block the data cache for two or more cycles, depending on whether a copy-back to main memory is required.

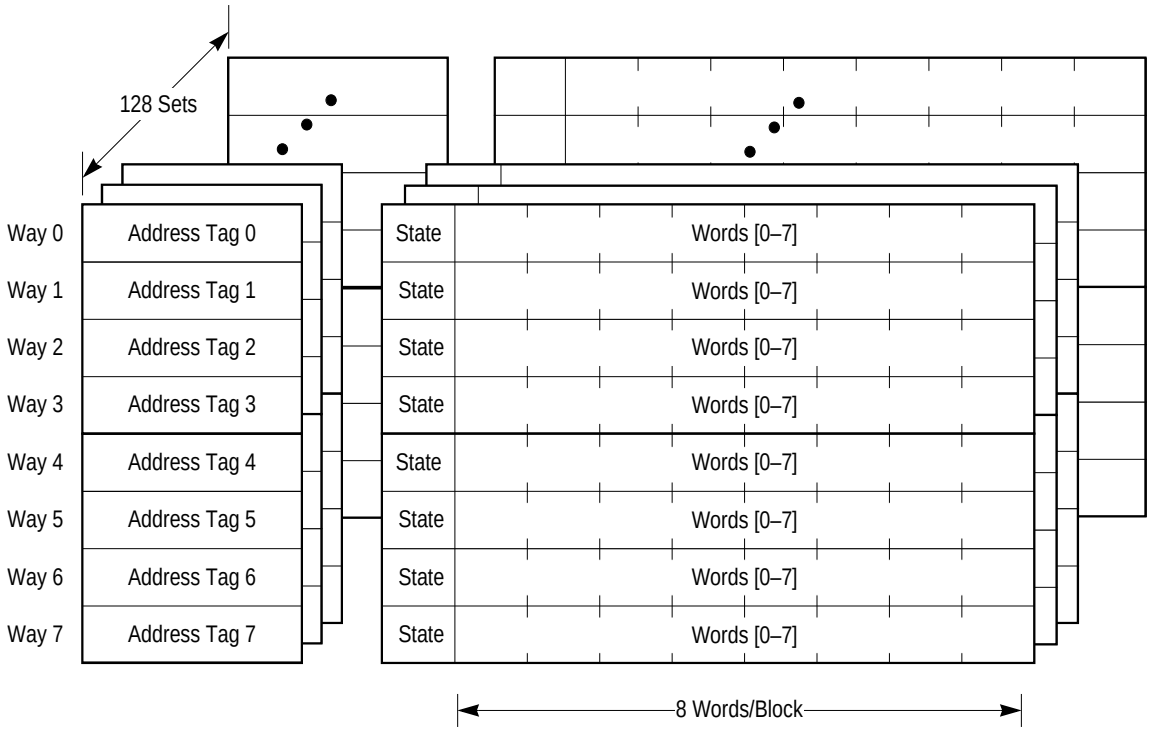


Figure 3-2. Data Cache Organization

3.2 Instruction Cache Organization

The instruction cache also consists of 128 sets of eight ways, as shown in Figure 3-3 on Page 3-5. Each way consists of 32 bytes, a single state bit, and an address tag. As with the data cache, each instruction cache block contains eight contiguous words from memory that are loaded from an eight-word boundary (that is, bits A[27–31] of the logical (effective) addresses are zero); as a result, cache blocks are aligned with page boundaries. Also, address bits A[20–26] provide the index to select a set, and bits A[27–29] select a word within a block.

The tags consist of bits PA[0–19]. Address translation occurs in parallel with set selection (from A[20–26]), and the higher order address bits (the tag bits in the cache) are physical.

The instruction cache differs from the data cache in that it does not implement MEI cache coherency protocol, and a single state bit is implemented that indicates only whether a cache block is valid or invalid. The instruction cache is not snooped, so if a processor modifies a memory location that may be contained in the instruction cache, software must ensure that such memory updates are visible to the instruction fetching mechanism. This can be achieved with the following instruction sequence:

```
dcbst    # update memory
sync     # wait for update
icbi     # remove (invalidate) copy in instruction cache
sync     # wait for ICBI operation to be globally performed
isync    # remove copy in own instruction buffer
```

These operations are necessary because the processor does not maintain instruction memory coherent with data memory. Software is responsible for enforcing coherency of instruction caches and data memory.

Since instruction fetching may bypass the data cache, changes made to items in the data cache may not be reflected in memory until after the instruction fetch completes.

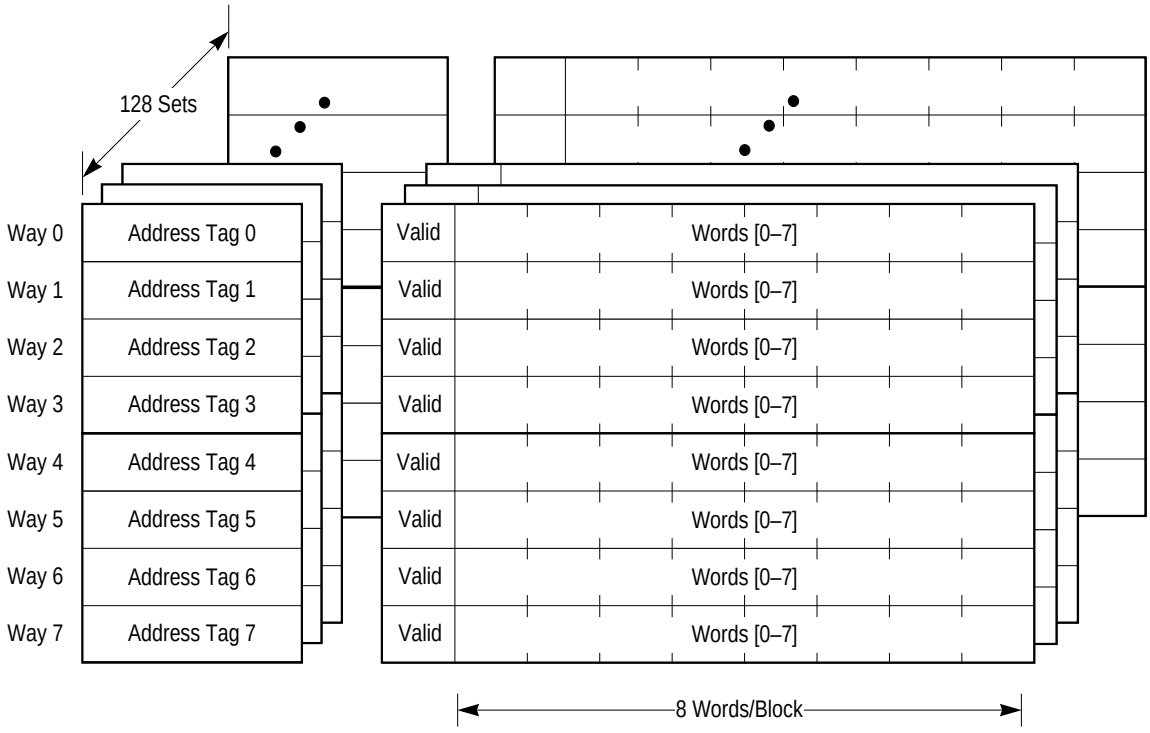


Figure 3-3. Instruction Cache Organization

3.3 Memory and Cache Coherency

The primary objective of a coherent memory system is to provide the same image of memory to all devices using the system. Coherency allows synchronization and cooperative use of shared resources. Otherwise, multiple copies of a memory location, some containing stale values, could exist in a system resulting in errors when the stale values are used. Each potential bus master must follow rules for managing the state of its cache. This section describes the coherency mechanisms of the PowerPC architecture and the three-state cache coherency protocol of Gekko's data cache.

Note that unless specifically noted, the discussion of coherency in this section applies to Gekko's data cache only. The instruction cache is not snooped. Instruction cache coherency must be maintained by software. However, Gekko does support a fast instruction cache invalidate capability as described in 3.4.1.4."

3.3.1 Memory/Cache Access Attributes (WIMG Bits)

Some memory characteristics can be set on either a block or page basis by using the WIMG bits in the BAT registers or page table entry (PTE), respectively. The WIMG attributes control the following functionality:

- Write-through (W bit)
- Caching-inhibited (I bit)
- Memory coherency (M bit)
- Guarded memory (G bit)

These bits allow both uniprocessor and multiprocessor system designs to exploit numerous system-level performance optimizations.

The WIMG attributes are programmed by the operating system for each page and block. The W and I attributes control how the processor performing an access uses its own cache. The M attribute ensures that coherency is maintained for all copies of the addressed memory location. The G attribute prevents out-of-order loading and prefetching from the addressed memory location.

The WIMG attributes occupy four bits in the BAT registers for block address translation and in the PTEs for page address translation. The WIMG bits are programmed as follows:

- The operating system uses the **mtspr** instruction to program the WIMG bits in the BAT registers for block address translation. The IBAT register pairs do not have a G bit and all accesses that use the IBAT register pairs are considered not guarded.
- The operating system writes the WIMG bits for each page into the PTEs in system memory as it sets up the page tables.

When an access requires coherency, the processor performing the access must inform the coherency mechanisms throughout the system that the access requires memory coherency. The M attribute determines the kind of access performed on the bus (global or local).

Software must exercise care with respect to the use of these bits if coherent memory support is desired. Careless specification of these bits may create situations that present coherency paradoxes to the processor. In particular, this can happen when the state of these bits is changed without appropriate precautions (such as flushing the pages that correspond to the changed bits from the caches of all processors in the system) or when the address translations of aliased real addresses specify different values for any of the WIMG bits. These coherency paradoxes can occur within a single processor or across several processors. It is important to note that in the presence of a paradox, the operating system software is responsible for correctness.

For real addressing mode (that is, for accesses performed with address translation disabled—MSR[IR] = 0 or MSR[DR] = 0 for instruction or data access, respectively), the WIMG bits are automatically generated as 0b0011 (the data is write-back, caching is enabled, memory coherency is enforced, and memory is guarded).

3.3.2 MEI Protocol

Gekko data cache coherency protocol is a coherent subset of the standard MESI four-state cache protocol that omits the shared state. Gekko's data cache characterizes each 32-byte block it contains as being in one of three MEI states. Addresses presented to the cache are indexed into the cache directory with bits A[20–26], and the upper-order 20 bits from the physical address translation (PA[0–19]) are compared against the indexed cache directory tags. If neither of the indexed tags matches, the result is a cache miss. If a tag matches, a cache hit occurred and the directory indicates the state of the cache block through two state bits kept with the tag. The three possible states for a cache block in the cache are the modified state (M), the exclusive state (E), and the invalid state (I).

The three MEI states are defined in Table 3-1.

Table 3-1. MEI State Definitions

MEI State	Definition
Modified (M)	The addressed cache block is present in the cache, and is modified with respect to system memory—that is, the modified data in the cache block has not been written back to memory. The cache block may be present in Gekko's L2 cache, but it is not present in any other coherent cache.
Exclusive (E)	The addressed cache block is present in the cache, and this cache has exclusive ownership of the addressed block. The addressed block may be present in Gekko's L2 cache, but it is not present in any other processor's cache. The data in this cache block is consistent with system memory.
Invalid (I)	This state indicates that the address block does not contain valid data or that the addressed cache block is not resident in the cache.

Gekko provides dedicated hardware to provide memory coherency by snooping bus transactions. Figure 3-4 on Page 3-8 shows the MEI cache coherency protocol, as enforced by Gekko. The information in this figure assumes that the WIM bits for the page or block are set to 001; that is, write-back, caching-not-inhibited, and memory coherency enforced. Since data cannot be shared, Gekko signals all cache block fills as if they were write misses (read-with-intent-to-modify), which flushes the corresponding copies of the data in all caches external to Gekko prior to the cache-block-fill operation. Following the cache block load, Gekko is the exclusive owner of the data and may write to it without a bus broadcast transaction. To maintain the three-state coherency, all global reads observed on the bus by Gekko are snooped as if they were writes, causing Gekko to flush the cache block (write the cache block back to memory and invalidate the cache block if it is modified, or simply invalidate the cache block if it is unmodified). The exception to this rule occurs when a snooped transaction is a caching-inhibited read (either burst or single-beat, where TT[0–4] = X1010; see Table 7-1 on Page 7-6 for clarification), in which case Gekko does not invalidate the snooped cache block. If the cache block is modified, the block is written back to memory, and the cache block is marked exclusive. If the cache block is marked exclusive, no bus action is taken, and the cache block remains in the exclusive state.

This treatment of caching-inhibited reads decreases the possibility of data thrashing by allowing noncaching devices to read data without invalidating the entry from Gekko's data cache.

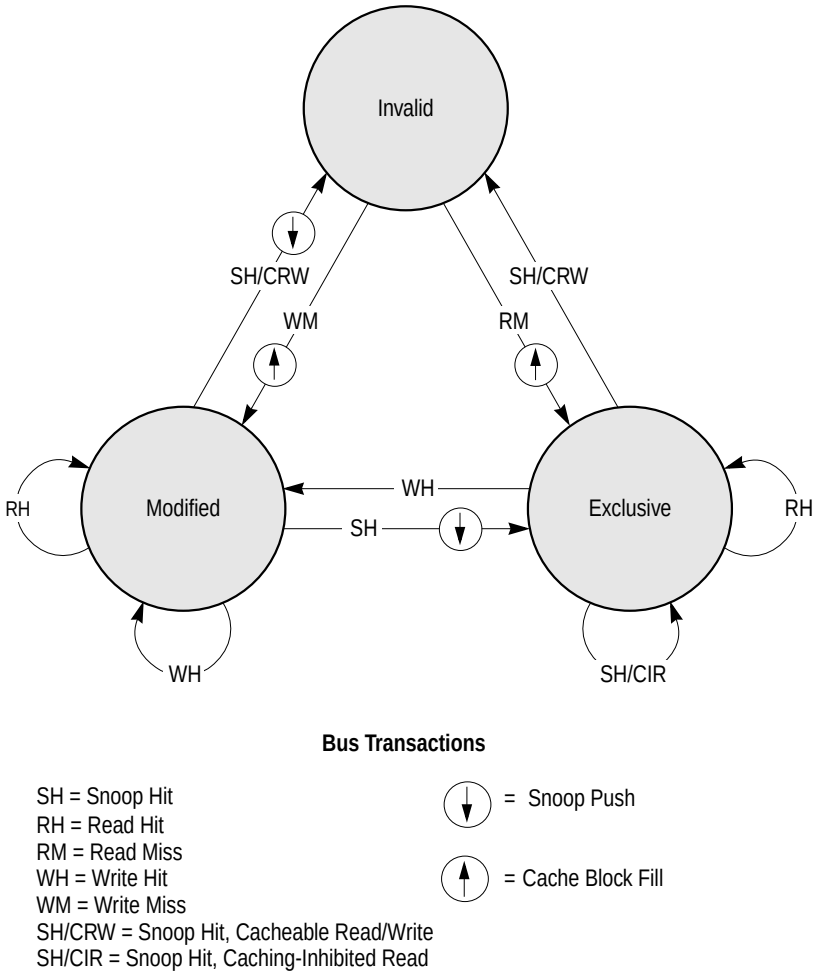


Figure 3-4. MEI Cache Coherency Protocol—State Diagram (WIM = 001)

Section 3.7 on Page 3-26 provides a detailed list of MEI transitions for various operations and WIM bit settings.

3.3.2.1 MEI Hardware Considerations

While Gekko provides the hardware required to monitor bus traffic for coherency, Gekko's data cache tags are single-ported, and a simultaneous load/store and snoop access represents a resource conflict. In general, the snoop access has highest priority and is given first access to the tags. The load or store access will then occur on the clock following the snoop. The snoop is not given priority into the tags when the snoop coincides with a tag write (for example, validation after a cache block load). In these situations, the snoop is retried and must re-arbitrate before the lookup is possible.

Occasionally, cache snoops cannot be serviced and must be retried. These retries occur if the cache is busy with a burst read or write when the snoop operation takes place.

Note that it is possible for a snoop to hit a modified cache block that is already in the process of being written to the copy-back buffer for replacement purposes. If this happens, Gekko retries the snoop, and raises the priority of the castout operation to allow it to go to the bus before the cache block fill.

Another consideration is page table aliasing. If a store hits to a modified cache block but the page table

entry is marked write-through (WIMG = 1xxx), then the page has probably been aliased through another page table entry which is marked write-back (WIMG = 0xxx). If this occurs, Gekko ignores the modified bit in the cache tag. The cache block is updated during the write-through operation and the block remains in the modified state.

The global ($\overline{GBL}$) signal, asserted as part of the address attribute field during a bus transaction, enables the snooping hardware of Gekko. Address bus masters assert $\overline{GBL}$ to indicate that the current transaction is a global access (that is, an access to memory shared by more than one device). If $\overline{GBL}$ is not asserted for the transaction, that transaction is not snooped by Gekko. Note that the $\overline{GBL}$ signal is not asserted for instruction fetches, and that $\overline{GBL}$ is asserted for all data read or write operations when using real addressing mode (that is, address translation is disabled).

Normally, $\overline{GBL}$ reflects the M-bit value specified for the memory reference in the corresponding translation descriptor(s). Care should be taken to minimize the number of pages marked as global, because the retry protocol enforces coherency and can use considerable bus bandwidth if much data is shared. Therefore, available bus bandwidth decreases as more memory is marked as global.

Gekko snoops a transaction if the transfer start ($\overline{TS}$) and $\overline{GBL}$ signals are asserted together in the same bus clock (this is a qualified snooping condition). No snoop update to Gekko cache occurs if the snooped transaction is not marked global. Also, because cache block castouts and snoop pushes do not require snooping, the $\overline{GBL}$ signal is not asserted for these operations.

When Gekko detects a qualified snoop condition, the address associated with the $\overline{TS}$ signal is compared with the cache tags. Snooping finishes if no hit is detected. If, however, the address hits in the cache, Gekko reacts according to the MEI protocol shown in Figure 3-4 on Page 3-8.

3.3.3 Coherency Precautions in Single Processor Systems

The following coherency paradoxes can be encountered within a single-processor system:

- Load or store to a caching-inhibited page (WIMG = x1xx) and a cache hit occurs.

  Gekko ignores any hits to a cache block in a memory space marked caching-inhibited (WIMG = x1xx). The access is performed on the external bus as if there were no hit. The data in the cache is not pushed, and the cache block is not invalidated.

- Store to a page marked write-through (WIMG = 1xxx) and a cache hit occurs to a modified cache block.

  Gekko ignores the modified bit in the cache tag. The cache block is updated during the write-through operation but the block remains in the modified state (M).

Note that when WIM bits are changed in the page tables or BAT registers, it is critical that the cache contents reflect the new WIM bit settings. For example, if a block or page that had allowed caching becomes caching-inhibited, software should ensure that the appropriate cache blocks are flushed to memory and invalidated.

3.3.4 Coherency Precautions in Multiprocessor Systems

Gekko's three-state coherency protocol permits no data sharing between Gekko and other caches. All burst reads initiated by Gekko are performed as read with intent to modify. Burst snoops are interpreted as read with intent to modify or read with no intent to cache. This effectively places all caches in the system into a three-state coherency scheme. Four-state caches may share data amongst themselves but not with Gekko.

3.3.5 Gekko-Initiated Load/Store Operations

Load and store operations are assumed to be weakly ordered on Gekko. The load/store unit (LSU) can perform load operations that occur later in the program ahead of store operations, even when the data cache is disabled (see 3.3.5.2). However, strongly ordered load and store operations can be enforced through the setting of the I bit (of the page WIMG bits) when address translation is enabled. Note that when address translation is disabled (real addressing mode), the default WIMG bits cause the I bit to be cleared (accesses are assumed to be cacheable), and thus the accesses are weakly ordered. Refer to Section 5.2 on Page 5-17 for a description of the WIMG bits when address translation is disabled.

Gekko does not provide support for direct-store segments. Operations attempting to access a direct-store segment will invoke a DSI exception. For additional information about DSI exceptions, refer to Section 4.5.3 on Page 4-17.

3.3.5.1 Performed Loads and Stores

The PowerPC architecture defines a performed load operation as one that has the addressed memory location bound to the target register of the load instruction. The architecture defines a performed store operation as one where the stored value is the value that any other processor will receive when executing a load operation (that is of course, until it is changed again). With respect to Gekko, caching-allowed (WIMG = x0xx) loads and caching-allowed, write-back (WIMG = 00xx) stores are performed when they have arbitrated to address the cache block. Note that in the event of a cache miss, these storage operations may place a memory request into the processor's memory queue, but such operations are considered an extension to the state of the cache with respect to snooping bus operations. Caching-inhibited (WIMG = x1xx) loads, caching-inhibited (WIMG = x1xx) stores, and write-through (WIMG = 1xxx) stores are performed when they have been successfully presented to the external 60x bus.

3.3.5.2 Sequential Consistency of Memory Accesses

The PowerPC architecture requires that all memory operations executed by a single processor be sequentially consistent with respect to that processor. This means that all memory accesses appear to be executed in program order with respect to exceptions and data dependencies.

Gekko achieves sequential consistency by operating a single pipeline to the cache/MMU. All memory accesses are presented to the MMU in exact program order and therefore exceptions are determined in order. Loads are allowed to bypass stores once exception checking has been performed for the store, but data dependency checking is handled in the load/store unit so that a load will not bypass a store with an address match. Note that although memory accesses that miss in the cache are forwarded to the memory queue for future arbitration for the external bus, all potential synchronous exceptions have been resolved before the cache. In addition, although subsequent memory accesses can address the cache, full coherency checking between the cache and the memory queue is provided to avoid dependency conflicts.

3.3.5.3 Atomic Memory References

The PowerPC architecture defines the Load Word and Reserve Indexed (**lwarx**) and the Store Word Conditional Indexed (**stwcx.**) instructions to provide an atomic update function for a single, aligned word of memory. These instructions can be used to develop a rich set of multiprocessor synchronization primitives.

NOTE: Atomic memory references constructed using **lwarx**/**stwcx.** instructions depend on the presence of a coherent memory system for correct operation. These instructions should not be expected to provide atomic access to noncoherent memory. For detailed information on these instructions, refer to Chapter 2, "Programming Model" and Chapter 12, "Instruction Set"” in this book.

The **lwarx** instruction performs a load word from memory operation and creates a reservation for the 32-byte section of memory that contains the accessed word. The reservation granularity is 32 bytes. The **lwarx** instruction makes a nonspecific reservation with respect to the executing processor and a specific reservation with respect to other masters. This means that any subsequent **stwcx.** executed by the same processor, regardless of address, will cancel the reservation. Also, any bus write or invalidate operation from another processor to an address that matches the reservation address will cancel the reservation.

The **stwcx.** instruction does not check the reservation for a matching address. The **stwcx.** instruction is only required to determine whether a reservation exists. The **stwcx.** instruction performs a store word operation only if the reservation exists. If the reservation has been cancelled for any reason, then the **stwcx.** instruction fails and clears the CR0[EQ] bit in the condition register. The architectural intent is to follow the **lwarx**/**stwcx.** instruction pair with a conditional branch which checks to see whether the **stwcx.** instruction failed.

If the page table entry is marked caching-allowed (WIMG = x0xx), and an **lwarx** access misses in the cache, then Gekko performs a cache block fill. If the page is marked caching-inhibited (WIMG = x1xx) or the cache is locked, and the access misses, then the **lwarx** instruction appears on the bus as a single-beat load. All bus operations that are a direct result of either an **lwarx** instruction or an **stwcx.** instruction are placed on the bus with a special encoding. Note that this does not force all **lwarx** instructions to generate bus transactions, but rather provides a means for identifying when an **lwarx** instruction does generate a bus transaction. If an implementation requires that all **lwarx** instructions generate bus transactions, then the associated pages should be marked as caching-inhibited.

Gekko’s data cache treats all **stwcx.** operations as write-through independent of the WIMG settings. However, if the **stwcx.** operation hits in Gekko’s L2 cache, then the operation completes with the reservation intact in the L2 cache. See Chapter 9, "L2 Cache, Locked D-Cache, DMA and Write Gather Pipe" for more information. Otherwise, the **stwcx.** operation continues to the bus interface unit for completion. When the write-through operation completes successfully, either in the L2 cache or on the 60x bus, then the data cache entry is updated (assuming it hits), and CR0[EQ] is modified to reflect the success of the operation. If the reservation is not intact, the **stwcx.** completes in the bus interface unit without performing a bus transaction, and without modifying either of the caches.

3.4 Cache Control

Gekko’s L1 caches are controlled by programming specific bits in the HID0 special-purpose register and by issuing dedicated cache control instructions. Section 3.4.1 describes the HID0 cache control bits, and Section 3.4.2 on Page 3-13 describes the cache control instructions.

3.4.1 Cache Control Parameters in HID0

The HID0 special-purpose register contains several bits that invalidate, disable, and lock the instruction and data caches. The following sections describe these facilities.

3.4.1.1 Data Cache Flash Invalidation

The data cache is automatically invalidated when Gekko is powered up and during a hard reset. However, a soft reset does not automatically invalidate the data cache. Software must use the HID0 data cache flash invalidate bit (HID0[DCFI]) if data cache invalidation is desired after a soft reset. Once HID0[DCFI] is set through an **mtspr** operation, Gekko automatically clears this bit in the next clock cycle (provided that the data cache is enabled in the HID0 register).

Note that some PowerPC microprocessors accomplish data cache flash invalidation by setting and clearing HID0[DCFI] with two consecutive **mtspr** instructions (that is, the bit is not automatically cleared by the microprocessor). Software that has this sequence of operations does not need to be changed to run on Gekko.

3.4.1.2 Data Cache Enabling/Disabling

The data cache may be enabled or disabled by using the data cache enable bit, HID0[DCE]. HID0[DCE] is cleared on power-up, disabling the data cache.

When the data cache is in the disabled state (HID0[DCE] = 0), the cache tag state bits are ignored, and all accesses are propagated to the L2 cache or 60x bus as single-beat transactions. Note that the $\overline{CI}$ (cache inhibit) signal always reflects the state of the caching-inhibited memory/cache access attribute (the I bit) independent of the state of HID0[DCE]. Also note that disabling the data cache does not affect the translation logic; translation for data accesses is controlled by MSR[DR].

The setting of the DCE bit must be preceded by a **sync** instruction to prevent the cache from being enabled or disabled in the middle of a data access. In addition, the cache must be globally flushed before it is disabled to prevent coherency problems when it is re-enabled.

Snooping is not performed when the data cache is disabled.

The **dcbz** instruction will cause an alignment exception when the data cache is disabled. The touch load (**dcbt** and **dcbtst**) instructions are no-ops when the data cache is disabled. Other cache operations (caused by the **dcbf**, **dcbst**, and **dcbi** instructions) are not affected by disabling the cache. This can potentially cause coherency errors. For example, a **dcbf** instruction that hits a modified cache block in the disabled cache will cause a copyback to memory of potentially stale data.

3.4.1.3 Data Cache Locking

The contents of the data cache can be locked by setting the data cache lock bit, HID0[DLOCK]. A data access that hits in a locked data cache is serviced by the cache. However, all accesses that miss in the locked cache are propagated to the L2 cache or 60x bus as single-beat transactions. Note that the $\overline{CI}$ signal always reflects the state of the caching-inhibited memory/cache access attribute (the I bit) independent of the state of HID0[DLOCK].

Gekko treats snoop hits to a locked data cache the same as snoop hits to an unlocked data cache. However, any cache block invalidated by a snoop hit remains invalid until the cache is unlocked.

The setting of the DLOCK bit must be preceded by a **sync** instruction to prevent the data cache from being locked during a data access.

3.4.1.4 Instruction Cache Flash Invalidation

The instruction cache is automatically invalidated when Gekko is powered up and during a hard reset. However, a soft reset does not automatically invalidate the instruction cache. Software must use the HID0 instruction cache flash invalidate bit (HID0[ICFI]) if instruction cache invalidation is desired after a soft reset. Once HID0[ICFI] is set through an **mtspr** operation, Gekko automatically clears this bit in the next clock cycle (provided that the instruction cache is enabled in the HID0 register).

NOTE: Some PowerPC microprocessors accomplish instruction cache flash invalidation by setting and clearing HID0[ICFI] with two consecutive **mtspr** instructions (that is, the bit is not automatically cleared by the microprocessor). Software that has this sequence of operations does not need to be changed to run on Gekko.

3.4.1.5 Instruction Cache Enabling/Disabling

The instruction cache may be enabled or disabled through the use of the instruction cache enable bit, HID0[ICE]. HID0[ICE] is cleared on power-up, disabling the instruction cache.

When the instruction cache is in the disabled state (HID[ICE] = 0), the cache tag state bits are ignored, and all instruction fetches are propagated to the L2 cache or 60x bus as single-beat transactions. Note that the $\overline{CI}$ signal always reflects the state of the caching-inhibited memory/cache access attribute (the I bit) independent of the state of HID0[ICE]. Also note that disabling the instruction cache does not affect the translation logic; translation for instruction accesses is controlled by MSR[IR].

The setting of the ICE bit must be preceded by an **isync** instruction to prevent the cache from being enabled or disabled in the middle of an instruction fetch. In addition, the cache must be globally flushed before it is disabled to prevent coherency problems when it is re-enabled. The **icbi** instruction is not affected by disabling the instruction cache.

3.4.1.6 Instruction Cache Locking

The contents of the instruction cache can be locked by setting the instruction cache lock bit, HID0[ILOCK]. An instruction fetch that hits in a locked instruction cache is serviced by the cache. However, all accesses that miss in the locked cache are propagated to the L2 cache or 60x bus as single-beat transactions. Note that the $\overline{CI}$ signal always reflects the state of the caching-inhibited memory/cache access attribute (the I bit) independent of the state of HID0[ILOCK].

The setting of the ILOCK bit must be preceded by an **isync** instruction to prevent the instruction cache from being locked during an instruction fetch.

3.4.2 Cache Control Instructions

The PowerPC architecture defines instructions for controlling both the instruction and data caches (when they exist). The cache control instructions, **dcbt**, **dcbtst**, **dcbz**, **dcbst**, **dcbf**, **dcbi**, and **icbi**, are intended for the management of the local L1 and L2 caches. Gekko interprets the cache control instructions as if they pertain only to its own L1 or L2 caches. These instructions are not intended for managing other caches in the system (except to the extent necessary to maintain coherency).

Gekko does not snoop cache control instruction broadcasts, except for **dcbz** when M = 1. The **dcbz** instruction is the only cache control instruction that causes a broadcast on the 60x bus (when M = 1) to maintain coherency. All other data cache control instructions (**dcbi**, **dcbf**, **dcbst** and **dcbz**) are not broadcast, unless broadcast is enabled through the HID0[ABE] configuration bit. Note that **dcbi**, **dcbf**, **dcbst** and **dcbz** do broadcast to Gekko's L2 cache, regardless of HID0[ABE]. The **icbi** instruction is never broadcast.

Gekko implements a new instruction, **dcbz_l**, to allocate lines in the locked cache when HID2[LCE] = 1. See Chapter 9, "L2 Cache, Locked D-Cache, DMA and Write Gather Pipe" for detail.

3.4.2.1 Data Cache Block Touch (dcbt) and Data Cache Block Touch for Store (dcbtst)

The Data Cache Block Touch (**dcbt**) and Data Cache Block Touch for Store (**dcbtst**) instructions provide potential system performance improvement through the use of software-initiated prefetch hints. Gekko treats these instructions identically (that is, a **dcbtst** instruction behaves exactly the same as a **dcbt** instruction on Gekko). Note that PowerPC implementations are not required to take any action based on the execution of these instructions, but they may choose to prefetch the cache block corresponding to the effective address into their cache.

Gekko loads the data into the cache when the address hits in the TLB or the BAT, is permitted load access from the addressed page, is not directed to a direct-store segment, and is directed at a cacheable page. Otherwise, Gekko treats these instructions as no-ops. The data brought into the cache as a result of this instruction is validated in the same manner that a load instruction would be (that is, it is marked as exclusive). The memory reference of a **dcbt** (or **dcbtst**) instruction causes the reference bit to be set. Note also that the successful execution of the **dcbt** (or **dcbtst**) instruction affects the state of the TLB and cache LRU bits as defined by the PLRU algorithm.

3.4.2.2 Data Cache Block Zero (dcbz)

The effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. The **dcbz** instruction is treated as a store to the addressed byte with respect to address translation and protection.

If the block containing the byte addressed by the EA is in the data cache, all bytes are cleared, and the tag is marked as modified (M). If the block containing the byte addressed by the EA is not in the data cache and the corresponding page is caching-allowed, the block is established in the data cache without fetching the block from main memory, and all bytes of the block are cleared, and the tag is marked as modified (M).

If the contents of the cache block are from a page marked memory coherence required (M = 1), an address-only bus transaction is run prior to clearing the cache block. The **dcbz** instruction is the only cache control instruction that causes a broadcast on the 60x bus (when M = 1) to maintain coherency. The other cache control instructions are not broadcast unless broadcasting is specifically enabled through the HID0[ABE] configuration bit. The **dcbz** instruction executes regardless of whether the cache is locked, but if the cache is disabled, an alignment exception is generated. If the page containing the byte addressed by the EA is caching-inhibited or write-through, then the system alignment exception handler is invoked. BAT and TLB protection violations generate DSI exceptions.

3.4.2.3 Data Cache Block Store (dcbst)

The effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. This instruction is treated as a load with respect to address translation and memory protection.

If the address hits in the cache and the cache block is in the exclusive (E) state, no action is taken. If the address hits in the cache and the cache block is in the modified (M) state, the modified block is written back to memory and the cache block is placed in the exclusive (E) state.

The execution of a **dcbst** instruction does not broadcast on the 60x bus unless broadcast is enabled through the HID0[ABE] bit. The function of this instruction is independent of the WIMG bit settings of the block containing the effective address. The **dcbst** instruction executes regardless of whether the cache is disabled or locked; however, a BAT or TLB protection violation generates a DSI exception.

3.4.2.4 Data Cache Block Flush (dcbf)

The effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. This instruction is treated as a load with respect to address translation and memory protection.

If the address hits in the cache, and the block is in the modified (M) state, the modified block is written back to memory and the cache block is placed in the invalid (I) state. If the address hits in the cache, and the cache block is in the exclusive (E) state, the cache block is placed in the invalid (I) state. If the address misses in the cache, no action is taken.

The execution of **dcbf** does not broadcast on the 60x bus unless broadcast is enabled through the HID0[ABE] bit. The function of this instruction is independent of the WIMG bit settings of the block containing the effective address. The **dcbf** instruction executes regardless of whether the cache is disabled or locked; however, a BAT or TLB protection violation generates a DSI exception.

3.4.2.5 Data Cache Block Invalidate (dcbi)

The effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. This instruction is treated as a store with respect to address translation and memory protection.

If the address hits in the cache, the cache block is placed in the invalid (I) state, regardless of whether the data is modified. Because this instruction may effectively destroy modified data, it is privileged (that is, **dcbi** is available to programs at the supervisor privilege level, MSR[PR] = 0). The execution of **dcbi** does not broadcast on the 60x bus unless broadcast is enabled through the HID0[ABE] bit. The function of this instruction is independent of the WIMG bit settings of the block containing the effective address. The **dcbi** instruction executes regardless of whether the cache is disabled or locked; however, a BAT or TLB protection violation generates a DSI exception.

3.4.2.6 Instruction Cache Block Invalidate (icbi)

For the **icbi** instruction, the effective address is not computed or translated, so it cannot generate a protection violation or exception. This instruction performs a virtual lookup into the instruction cache (index only). All ways of the selected instruction cache set are invalidated.

The **icbi** instruction is not broadcast on the 60x bus. The **icbi** instruction invalidates the cache blocks independent of whether the cache is disabled or locked.

3.5 Cache Operations

This section describes Gekko cache operations.

3.5.1 Cache Block Replacement/Castout Operations

Both the instruction and data cache use a pseudo least-recently-used (PLRU) replacement algorithm when a new block needs to be placed in the cache. When the data to be replaced is in the modified (M) state, that data is written into a castout buffer while the missed data is being accessed on the bus. When the load completes, Gekko then pushes the replaced cache block from the castout buffer to the L2 cache (if L2 is enabled) or to main memory (if L2 is disabled).

The replacement logic first checks to see if there are any invalid blocks in the set and chooses the lowest-order, invalid block (L[0–7]) as the replacement target. If all eight blocks in the set are valid, the PLRU algorithm is used to determine which block should be replaced. The PLRU algorithm is shown in Figure 3-5 on Page 3-16.

Each cache is organized as eight blocks per set by 128 sets. There is a valid bit for each block in

the cache, L[0–7]. When all eight blocks in the set are valid, the PLRU algorithm is used to select the replacement target. There are seven PLRU bits, B[0–6] for each set in the cache. For every hit in the cache, the PLRU bits are updated using the rules specified in Table 3-2 on Page 3-17.

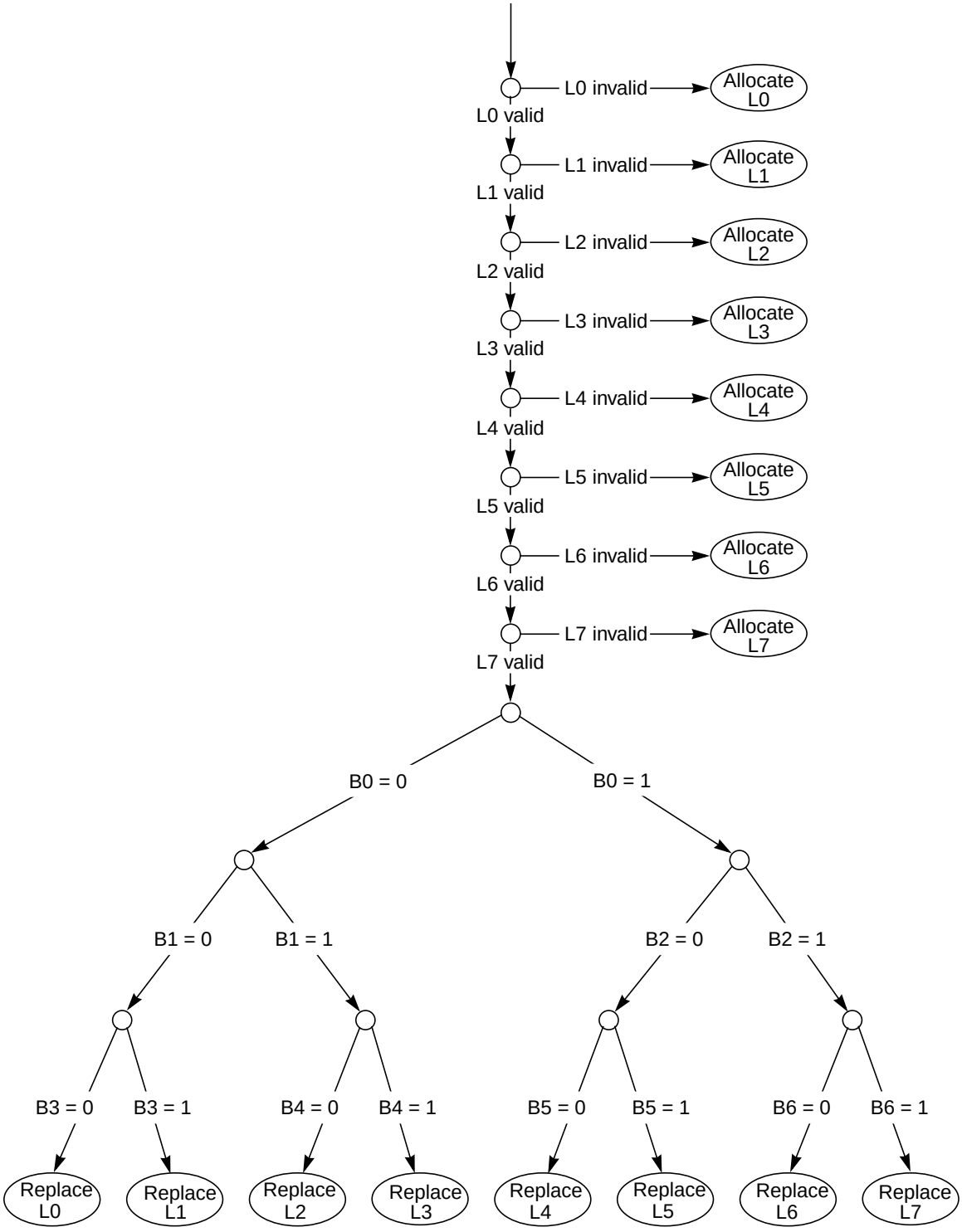


Figure 3-5. PLRU Replacement Algorithm

Table 3-2. PLRU Bit Update Rules

If the Current Access is To:	Then the PLRU bits are Changed to:[1]						
	B0	B1	B2	B3	B4	B5	B6
L0	1	1	x	1	x	x	x
L1	1	1	x	0	x	x	x
L2	1	0	x	x	1	x	x
L3	1	0	x	x	0	x	x
L4	0	x	1	x	x	1	x
L5	0	x	1	x	x	0	x
L6	0	x	0	x	x	x	1
L7	0	x	0	x	x	x	0

Note: [1]x = Does not change

If all eight blocks are valid, then a block is selected for replacement according to the PLRU bit encodings shown in Table 3-3.

Table 3-3. PLRU Replacement Block Selection

If the PLRU Bits Are:						Then the Block Selected for Replacement Is:
B0	0	B1	0	B3	0	L0
	0		0		1	L1
	0		1	B4	0	L2
	0		1		1	L3
	1	B2	0	B5	0	L4
	1		0		1	L5
	1		1	B6	0	L6
	1		1		1	L7

During power-up or hard reset, all the valid bits of the blocks are cleared and the PLRU bits cleared to point to block L0 of each set. Note that this is also the state of the data or instruction cache after setting their respective flash invalidate bit (HID0[DCFI] or HID0[ICFI]).

3.5.2 Cache Flush Operations

The instruction cache can be invalidated by executing a series of **icbi** instructions or by setting HID0[ICFI]. The data cache can be invalidated by executing a series of **dcbi** instructions or by setting HID0[DCFI].

Any modified entries in the data cache can be copied back to memory (flushed) by using the **dcbf** instruction or by executing a series of 12 uniquely addressed load or **dcbz** instructions to each of the 128 sets. The address space should not be shared with any other process to prevent snoop hit invalidations during the flushing routine. Exceptions should be disabled during this time so that the PLRU algorithm does not get disturbed.

The data cache flush assist bit, HID0[DCFA], simplifies the software flushing process. When set, HID0[DCFA] forces the PLRU replacement algorithm to ignore the invalid entries and follow the replacement sequence defined by the PLRU bits. This reduces the series of uniquely addressed load or **dcbz** instructions to eight per set. HID0[DCFA] should be set just prior to the beginning of the cache flush routine and cleared after the series of instructions is complete.

3.5.3 Data Cache-Block-Fill Operations

Gekko's data cache blocks are filled in four beats of 64 bits each, with the critical double word loaded first. The data cache is not blocked to internal accesses while the load (caused by a cache miss) completes. This functionality is sometimes referred to as 'hits under misses,' because the cache can service a hit while a cache miss fill is waiting to complete. The critical-double-word read from memory is simultaneously written to the data cache and forwarded to the requesting unit, thus minimizing stalls due to cache fill latency.

A cache block is filled after a read miss or write miss (read-with-intent-to-modify) occurs in the cache. The cache block that corresponds to the missed address is updated by a burst transfer of the data from the L2 or system memory. Note that if a read miss occurs in a system with multiple bus masters, and the data is modified in another cache, the modified data is first written to external memory before the cache fill occurs.

3.5.4 Instruction Cache-Block-Fill Operations

Gekko's instruction cache blocks are loaded in four beats of 64 bits each, with the critical double word loaded first. The instruction cache is not blocked to internal accesses while the fetch (caused by a cache miss) completes. On a cache miss, the critical and following double words read from memory are simultaneously written to the instruction cache and forwarded to the instruction queue, thus minimizing stalls due to cache fill latency. There is no snooping of the instruction cache.

3.5.5 Data Cache-Block-Push Operation

When a cache block in Gekko is snooped and hit by another bus master and the data is modified, the cache block must be written to memory and made available to the snooping device. The cache block is said to be pushed out onto the 60x bus.

3.6 L1 Caches and 60x Bus Transactions

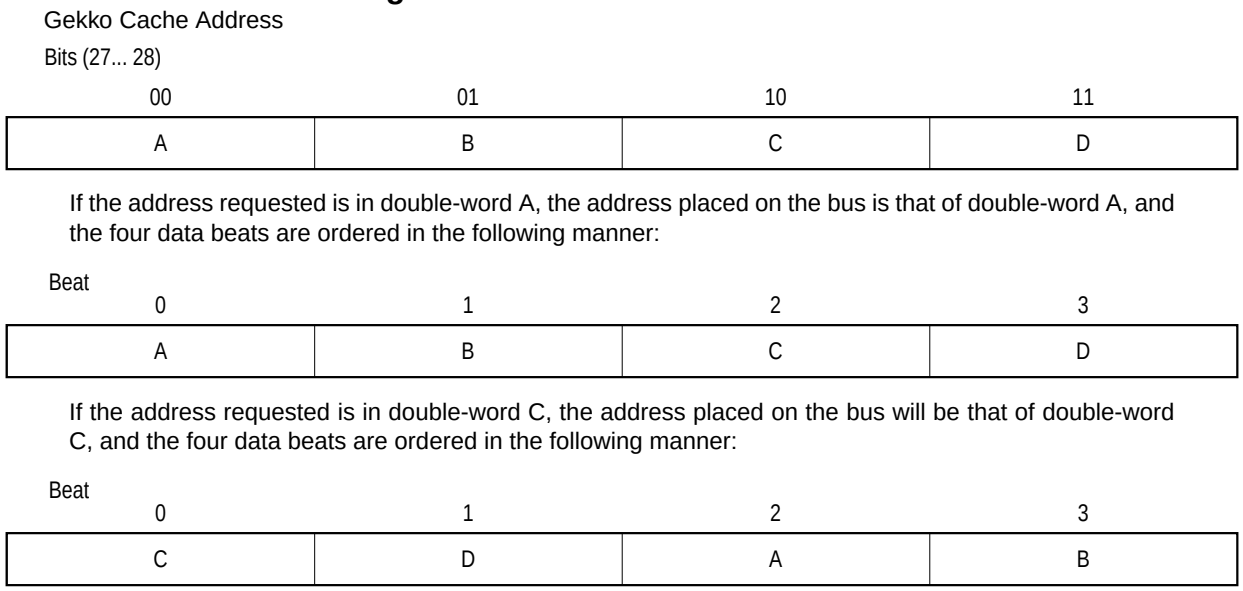
Gekko transfers data to and from the cache in single-beat transactions of two words, or in four-beat transactions of eight words which fill a cache block. Single-beat bus transactions can transfer from one to eight bytes to or from Gekko, and can be misaligned. Single-beat transactions can be caused by cache write-through accesses, caching-inhibited accesses (WIMG = x1xx), accesses when the cache is disabled (HID0[DCE] bit is cleared), or accesses when the cache is locked (HID0[DLOCK] bit is cleared).

Burst transactions on Gekko always transfer eight words of data at a time, and are aligned to a double-word boundary. Gekko transfer burst ($\overline{\text{TBST}}$) output signal indicates to the system whether

the current transaction is a single-beat transaction or four-beat burst transfer. Burst transactions have an assumed address order. For cacheable read operations, instruction fetches, or cacheable, non-write-through write operations that miss the cache, Gekko presents the double-word-aligned address associated with the load/store instruction or instruction fetch that initiated the transaction.

As shown in Figure 3-6, the first quad word contains the address of the load/store or instruction fetch that missed the cache. This minimizes latency by allowing the critical code or data to be forwarded to the processor before the rest of the block is filled. For all other burst operations, however, the entire block is transferred in order (oct-word-aligned). Critical-double-word-first fetching on a cache miss applies to both the data and instruction cache.

Figure 3-6 Gekko Cache Addresses



3.6.1 Read Operations and the MEI Protocol

The MEI coherency protocol affects how Gekko data cache performs read operations on the 60x bus. All reads (except for caching-inhibited reads) are encoded on the bus as read-with-intent-to-modify (RWITM) to force flushing of the addressed cache block from other caches in the system.

The MEI coherency protocol also affects how Gekko snoops read operations on the 60x bus. All reads snooped from the 60x bus (except for caching-inhibited reads) are interpreted as RWITM to cause flushing from Gekko's cache. Single-beat reads ($\overline{\text{TBST}}$ negated) are interpreted by Gekko as caching inhibited.

These actions for read operations allow Gekko to operate successfully (coherently) on the bus with other bus masters that implement either the three-state MEI or a four-state MESI cache coherency protocol.

3.6.2 Bus Operations Caused by Cache Control Instructions

The cache control, TLB management, and synchronization instructions supported by Gekko may affect or be affected by the operation of the 60x bus. The operation of the instructions may also indirectly cause bus transactions to be performed, or their completion may be linked to the bus.

The **dcbz** instruction is the only cache control instruction that causes an address-only broadcast on the 60x bus. All other data cache control instructions (**dcbi**, **dcbf**, **dcbst**, and **dcbz**) are not broadcast unless specifically enabled through the HID0[ABE] configuration bit. Note that **dcbi**,

dcbf, **dcbst**, and **dcbz** do broadcast to Gekko's L2 cache, regardless of HID0[ABE]. HID0[ABE] also controls the broadcast of the **sync** and **eieio** instructions.

The **icbi** instruction is never broadcast. No broadcasts by other masters are snooped by Gekko (except for **dcbz** kill block transactions). The **dcbz_l** instruction is never broadcast. For detailed information on the cache control instructions, refer to Chapter 2, "Programming Model" and Chapter 12, "Instruction Set" in this book.

Table 3-4 provides an overview of the bus operations initiated by cache control instructions. Note that the information in this table assumes that the WIM bits are set to 001; that is, the cache is operating in write-back mode, caching is permitted and coherency is enforced.

Table 3-4. Bus Operations Caused by Cache Control Instructions (WIM = 001)

Instruction	Current Cache State	Next Cache State	Bus Operation	Comment
sync	Don't care	No change	**sync** (if enabled in HID0[ABE])	Waits for memory queues to complete bus activity
tlbie	—	—	None	—
tlbsync	—	—	None	Waits for the negation of the $\overline{\text{TLBSYNC}}$ input signal to complete
eieio	Don't care	No change	**eieio** (if enabled in HID0[ABE])	Address-only bus operation
icbi	Don't care	I	None	—
dcbi	Don't care	I	Kill block (if enabled in HID0[ABE])	Address-only bus operation
dcbf	I, E	I	Flush block (if enabled in HID0[ABE])	Address-only bus operation
dcbf	M	I	Write with kill	Block is pushed
dcbst	I, E	No change	Clean block (if enabled in HID0[ABE])	Address-only bus operation
dcbst	M	E	Write with kill	Block is pushed
dcbz	I	M	Write with kill	—
dcbz	E, M	M	Kill block	Writes over modified data
dcbz_l	M, E, I	M	None	—
dcbt	I	E	Read-with-intent-to-modify	Fetched cache block is stored in the cache
dcbt	E, M	No change	None	—
dcbtst	I	E	Read-with-intent-to-modify	Fetched cache block is stored in the cache
dcbtst	E,M	No change	None	—

For additional details about the specific bus operations performed by Gekko, see Chapter 8, "Bus Interface Operation" in this manual.

3.6.3 Snooping

Gekko maintains data cache coherency in hardware by coordinating activity between the data cache, the bus interface logic, the L2 cache, and the memory system. Gekko has a copy-back cache which relies on bus snooping to maintain cache coherency with other caches in the system. For Gekko, the coherency size of the bus is the size of a cache block, 32 bytes. This means that any bus transactions that cross an aligned 32-byte boundary must present a new address onto the bus at that boundary for proper snoop operation by Gekko, or they must operate noncoherently with respect to Gekko.

As bus operations are performed on the bus by other bus masters, Gekko's bus snooping logic monitors the addresses and transfer attributes that are referenced. Gekko snoops the bus transactions during the cycle that $\overline{TS}$ is asserted for any of the following qualified snoop conditions:

- The global signal ($\overline{GBL}$) is asserted indicating that coherency enforcement is required.
- A reservation is currently active in Gekko as the result of an **lwarx** instruction, and the transfer type attributes (TT[0–4]) indicate a write or kill operation. These transactions are snooped regardless of whether $\overline{GBL}$ is asserted to support reservations in the MEI cache protocol.

All transactions snooped by Gekko are checked for correct address bus parity. Every assertion of $\overline{TS}$ detected by Gekko (whether snooped or not) must be followed by an accompanying assertion of $\overline{AACK}$.

The locked cache and bus snoop are incompatible. HID2[LCE] shall be kept at 0 for systems which generate snoop transactions

Once a qualified snoop condition is detected on the bus, the snooped address associated with $\overline{TS}$ is compared against the data cache tags, memory queues, and/or other storage elements as appropriate. The L1 data cache tags and L2 cache tags are snooped for standard data cache coherency support. No snooping is done in the instruction cache for coherency.

The memory queues are snooped for pipeline collisions and memory coherency collisions. A pipeline collision is detected when another bus master addresses any portion of a line that this 750's data cache is currently in the process of loading (L1 loading from L2, or L1/L2 loading from memory). A memory coherency collision occurs when another bus master addresses any portion of a line that Gekko has currently queued to write to memory from the data cache (castout or copy-back), but has not yet been granted bus access to perform.

If a snooped transaction results in a cache hit or pipeline collision or memory queue collision, Gekko asserts $\overline{ARTRY}$ on the 60x bus. The current bus master, detecting the assertion of the $\overline{ARTRY}$ signal, should abort the transaction and retry it at a later time, so that Gekko can first perform a write operation back to memory from its cache or memory queues. Gekko may also retry a bus transaction if it is unable to snoop the transaction on that cycle due to internal resource conflicts. Additional snoop action may be forwarded to the cache as a result of a snoop hit in some cases (a cache push of modified data, or a cache block invalidation). There is no immediate way for another CPU bus agent to determine the cause of Gekko $\overline{ARTRY}$.

Implementation Note: Snooping of the memory queues for pipeline collisions, as described above, is performed for burst read operations in progress only. In this case, the read address has completed on the bus, however, the data tenure may be either in-progress or not yet started by the processor. During this time Gekko will retry any other global access to that line by another bus master until all data has been received in it's L1 cache. Pipeline collisions, however, do not apply for burst write operations in progress. If Gekko has completed an address tenure for a burst write, and is currently waiting for a data bus grant or is currently transferring data to memory, it will not generate an address retry to another bus master that addresses the line. It is the responsibility of the

memory system to handle this collision (usually by keeping the data transactions to memory in order). Note also that all burst writes by Gekko and 603e are performed as non-global, and hence do not normally enable snooping, even for address collision purposes. (Snooping may still occur for reservation cancelling purposes.)

3.6.4 Snoop Response to 60x Bus Transactions

There are several bus transaction types defined for the 60x bus. The transactions in Table 3-5 correspond to the transfer type signals TT[0–4], which are described in Section 7.2.4.1 on Page 7-6. Gekko never retries a transaction in which $\overline{\text{GBL}}$ is not asserted, even if the tags are busy or there is a tag hit. Reservations are snooped regardless of the state of $\overline{\text{GBL}}$.

Table 3-5. Response to Snooped Bus Transactions

Snooped Transaction	TT[0–4]	Gekko Response
Clean block	00000	No action is taken.
Flush block	00100	No action is taken.
SYNC	01000	No action is taken.
Kill block	01100	The kill block operation is an address-only bus transaction initiated when a **dcbz** or **dcbi** instruction is executed • If the addressed cache block is in the exclusive (E) state, the cache block is placed in the invalid (I) state. • If the addressed cache block is in the modified (M) state, Gekko asserts $\overline{\text{ARTRY}}$ and initiates a push of the modified block out of the cache and the cache block is placed in the invalid (I) state. • If the address misses in the cache, no action is taken. Any reservation associated with the address is canceled.
EIEIO	10000	No action is taken.
External control word write	10100	No action is taken.
TLB invalidate	11000	No action is taken.
External control word read	11100	No action is taken.
lwarx reservation set	00001	No action is taken.
Reserved	00101	—
TLBSYNC	01001	No action is taken.
ICBI	01101	No action is taken.
Reserved	1XX01	—

Table 3-5. Response to Snooped Bus Transactions (Continued)

Snooped Transaction	TT[0–4]	Gekko Response
Write-with-flush	00010	A write-with-flush operation is a single-beat or burst transaction initiated when a caching-inhibited or write-through store instruction is executed. • If the addressed cache block is in the exclusive (E) state, the cache block is placed in the invalid (I) state. • If the addressed cache block is in the modified (M) state, Gekko asserts $\overline{\text{ARTRY}}$ and initiates a push of the modified block out of the cache and the cache block is placed in the invalid (I) state. • If the address misses in the cache, no action is taken. Any reservation associated with the address is canceled.
Write-with-kill	00110	A write-with-kill operation is a burst transaction initiated due to a castout, caching-allowed push, or snoop copy -back. • If the address hits in the cache, the cache block is placed in the invalid (I) state (killing modified data that may have been in the block). • If the address misses in the cache, no action is taken. Any reservation associated with the address is canceled.
Read	01010	A read operation is used by most single-beat and burst load transactions on the bus. For single-beat, caching-inhibited read transaction: • If the addressed cache block is in the exclusive (E) state, the cache block remains in the exclusive (E) state. • If the addressed cache block is in the modified (M) state, Gekko asserts $\overline{\text{ARTRY}}$ and initiates a push of the modified block out of the cache and the cache block is placed in the exclusive (E) state. • If the address misses in the cache, no action is taken. For burst read transactions: • If the addressed cache block is in the exclusive (E) state, the cache block is placed in the invalid (I) state. • If the addressed cache block is in the modified (M) state, Gekko asserts $\overline{\text{ARTRY}}$ and initiates a push of the modified block out of the cache and the cache block is placed in the invalid (I) state. • If the address misses in the cache, no action is taken.
Read-with-intent-to-modify (RWITM)	01110	A RWITM operation is issued to acquire exclusive use of a memory location for the purpose of modifying it. • If the addressed cache block is in the exclusive (E) state, the cache block is placed in the invalid (I) state. • If the addressed cache block is in the modified (M) state, Gekko asserts $\overline{\text{ARTRY}}$ and initiates a push of the modified block out of the cache and the cache block is placed in the invalid (I) state. • If the address misses in the cache, no action is taken.
Write-with-flush-atomic	10010	Write-with-flush-atomic operations occur after the processor issues an **stwcx.** instruction. • If the addressed cache block is in the exclusive (E) state, the cache block is placed in the invalid (I) state. • If the addressed cache block is in the modified (M) state, Gekko asserts $\overline{\text{ARTRY}}$ and initiates a push of the modified block out of the cache and the cache block is placed in the invalid (I) state. • If the address misses in the cache, no action is taken. Any reservation is canceled, regardless of the address.
Reserved	10110	—

Table 3-5. Response to Snooped Bus Transactions (Continued)

Snooped Transaction	TT[0–4]	Gekko Response
Read-atomic	11010	Read atomic operations appear on the bus in response to **lwarx** instructions and generate the same snooping responses as read operations.
Read-with-intent-to-modify-atomic	11110	The RWITM atomic operations appear on the bus in response to **stwcx.** instructions and generate the same snooping responses as RWITM operations.
Reserved	00011	—
Reserved	00111	—
Read-with-no-intent-to-cache (RWNITC)	01011	A RWNITC operation is issued to acquire exclusive use of a memory location with no intention of modifying the location. • If the addressed cache block is in the exclusive (E) state, the cache block remains in the exclusive (E) state. • If the addressed cache block is in the modified (M) state, Gekko asserts $\overline{\text{ARTRY}}$ and initiates a push of the modified block out of the cache and the cache block is placed in the exclusive (E) state. • If the address misses in the cache, no action is taken.
Reserved	01111	—
Reserved	1XX11	—

3.6.5 Transfer Attributes

In addition to the address and transfer type signals, Gekko supports the transfer attribute signals $\overline{\text{TBST}}$, TSIZ[0–2], $\overline{\text{WT}}$, $\overline{\text{CI}}$, and $\overline{\text{GBL}}$. The TBST and TSIZ[0–2] signals indicate the data transfer size for the bus transaction.

The $\overline{\text{WT}}$ signal reflects the write-through status (the complement of the W bit) for the transaction as determined by the MMU address translation during write operations. $\overline{\text{WT}}$ is asserted for burst writes due to **dcbf** (flush) and **dcbst** (clean) instructions, and for snoop pushes; $\overline{\text{WT}}$ is negated for **ecowx** transactions. Since the write-through status is not meaningful for reads, Gekko uses the $\overline{\text{WT}}$ signal during read transactions to indicate that the transaction is an instruction fetch ($\overline{\text{WT}}$ negated), or not an instruction fetch ($\overline{\text{WT}}$ asserted).

The $\overline{\text{CI}}$ signal reflects the caching-inhibited/allowed status (the complement of the I bit) of the transaction as determined by the MMU address translation even if the L1 caches are disabled or locked. $\overline{\text{CI}}$ is always asserted for **eciwx**/**ecowx** bus transactions independent of the address translation.

The $\overline{\text{GBL}}$ signal reflects the memory coherency requirements (the complement of the M bit) of the transaction as determined by the MMU address translation. Castout and snoop copy-back operations (TT[0–4] = 00110) are generally marked as nonglobal ($\overline{\text{GBL}}$ negated) and are not snooped (except for reservation monitoring). Other masters, however, may perform DMA write operations with this encoding but marked global ($\overline{\text{GBL}}$ asserted) and thus must be snooped. Table 3-6 summarizes the address and transfer attribute information presented on the bus by Gekko for various master or snoop-related transactions.

Table 3-6. Address/Transfer Attribute Summary

Bus Transaction	A[0–31]	TT[0–4]	$\overline{\text{TBST}}$	TSIZ[0–2]	$\overline{\text{GBL}}$	$\overline{\text{WT}}$	$\overline{\text{CI}}$
Instruction fetch operations:							
Burst (caching-allowed)	PA[0–28] || 0b000	0 1 1 1 0	0	0 1 0	¬ M	1	1*
Single-beat read (caching-inhibited or cache disabled)	PA[0–28] || 0b000	0 1 0 1 0	1	0 0 0	¬ M	1	¬ I
Data cache operations:							
Cache block fill (due to load or store miss)	PA[0–28] || 0b000	A 1 1 1 0	0	0 1 0	¬ M	0	1*
Castout (normal replacement)	CA[0–26] || 0b00000	0 0 1 1 0	0	0 1 0	1	1	1*
Push (cache block push due to **dcbf**/**dcbst**)	PA[0–26] || 0b00000	0 0 1 1 0	0	0 1 0	1	0	1*
Snoop copyback	CA[0–26] || 0b00000	0 0 1 1 0	0	0 1 0	1	0	1*
Data cache bypass operations:							
Single-beat read (caching-inhibited or cache disabled)	PA[0–31]	A 1 0 1 0	1	S S S	¬ M	0	¬ I
Single-beat write (caching-inhibited, write-through, or cache disabled)	PA[0–31]	0 0 0 1 0	1	S S S	¬ M	¬W	¬ I
Special instructions:							
dcbz (addr-only)	PA[0–28] || 0b000	0 1 1 0 0	0	0 1 0	0*	0	1*
dcbi (if HID0[ABE] = 1, addr-only)	PA[0–26] || 0b00000	0 1 1 0 0	0	0 1 0	¬ M	0	1*
dcbf (if HID0[ABE] = 1, addr-only)	PA[0–26] || 0b00000	0 0 1 0 0	0	0 1 0	¬ M	0	1*
dcbst (if HID0[ABE] = 1, addr-only)	PA[0–26] || 0b00000	0 0 0 0 0	0	0 1 0	¬ M	0	1*
sync (if HID0[ABE] = 1, addr-only)	0x0000_0000	0 1 0 0 0	0	0 1 0	0	0	0
eieio (if HID0[ABE] = 1, addr-only)	0x0000_0000	1 0 0 0 0	0	0 1 0	0	0	0
stwcx. (always single-beat write)	PA[0–29] || 0b00	1 0 0 1 0	1	1 0 0	¬ M	¬ W	¬ I
eciwx	PA[0–29] || 0b00	1 1 1 0 0	EAR[28–31]		1	0	0
ecowx	PA[0–29] || 0b00	1 0 1 0 0	EAR[28–31]		1	1	0

Notes:
PA = Physical address, CA = Cache address.
W,I,M = WIM state from address translation; ¬ = complement; 0*or 1* = WIM state implied by transaction type in table
For instruction fetches, reflection of the M bit must be enabled through HID0[IFEM].
A = Atomic; high if **lwarx**, low otherwise
S = Transfer size
Special instructions listed may not generate bus transactions depending on cache state.

3.7 MEI State Transactions

Table 3-7 shows MEI state transitions for various operations. Bus operations are described in Table 3-4 on Page 3-20.

Table 3-7. MEI State Transitions

Operation	Cache Operation	Bus sync	WIM	Current Cache State	Next Cache State	Cache Actions	Bus Operation
Load (T = 0)	Read	No	x0x	I	Same	1 Cast out of modified block (as required)	Write-with-kill
						2 Pass four-beat read to memory queue	Read
Load (T = 0)	Read	No	x0x	E,M	Same	Read data from cache	—
Load (T = 0)	Read	No	x1x	I	Same	Pass single-beat read to memory queue	Read
Load (T = 0)	Read	No	x1x	E	I	CRTRY read	—
Load (T = 0)	Read	No	x1x	M	I	CRTRY read (push sector to write queue)	Write-with-kill
lwarx	Read	Acts like other reads but bus operation uses special encoding					
Store (T = 0)	Write	No	00x	I	Same	Cast out of modified block (if necessary)	Write-with-kill
						Pass RWITM to memory queue	RWITM
Store (T = 0)	Write	No	00x	E,M	M	Write data to cache	—
Store **stwcx.** (T = 0)	Write	No	10x	I	Same	Pass single-beat write to memory queue	Write-with-flush
Store **stwcx.** (T = 0)	Write	No	10x	E	Same	Write data to cache	—
						Pass single-beat write to memory queue	Write-with-flush
Store **stwcx.** (T = 0)	Write	No	10x	M	Same	CRTRY write	—
						Push block to write queue	Write-with-kill
Store (T = 0) or **stwcx.** (WIM = 10x)	Write	No	x1x	I	Same	Pass single-beat write to memory queue	Write-with-flush
Store (T = 0) or **stwcx.** (WIM = 10x)	Write	No	x1x	E	I	CRTRY write	—

Table 3-7. MEI State Transitions (Continued)

Operation	Cache Operation	Bus sync	WIM	Current Cache State	Next Cache State	Cache Actions	Bus Operation
Store (T = 0) or **stwcx.** (WIM = 10x)	Write	No	x1x	M	I	CRTRY write	—
						Push block to write queue	Write-with-kill
stwcx.	Conditional write	If the reserved bit is set, this operation is like other writes except the bus operation uses a special encoding.					
dcbf	Data cache block flush	No	xxx	I,E	Same	CRTRY **dcbf**	—
						Pass flush	Flush
				Same	I	State change only	—
dcbf	Data cache block flush	No	xxx	M	I	Push block to write queue	Write-with-kill
dcbst	Data cache block store	No	xxx	I,E	Same	CRTRY **dcbst**	—
						Pass clean	Clean
				Same	Same	No action	—
dcbst	Data cache block store	No	xxx	M	E	Push block to write queue	Write-with-kill
dcbz	Data cache block set to zero	No	x1x	x	x	Alignment trap	—
dcbz	Data cache block set to zero	No	10x	x	x	Alignment trap	—
dcbz	Data cache block set to zero	Yes	00x	I	Same	CRTRY **dcbz**	—
						Cast out of modified block	Write-with-kill
						Pass kill	Kill
				Same	M	Clear block	—
dcbz	Data cache block set to zero	No	00x	E,M	M	Clear block	—
dcbt	Data cache block touch	No	x1x	I	Same	Pass single-beat read to memory queue	Read
dcbt	Data cache block touch	No	x1x	E	I	CRTRY read	—
dcbt	Data cache block touch	No	x1x	M	I	CRTRY read	—
						Push block to write queue	Write-with-kill

Table 3-7. MEI State Transitions (Continued)

Operation	Cache Operation	Bus sync	WIM	Current Cache State	Next Cache State	Cache Actions	Bus Operation
dcbt	Data cache block touch	No	x0x	I	Same	Cast out of modified block (as required)	Write-with-kill
						Pass four-beat read to memory queue	Read
dcbt	Data cache block touch	No	x0x	E,M	Same	No action	—
Single-beat read	Reload dump 1	No	xxx	I	Same	Forward data_in	—
Four-beat read (double-word-aligned)	Reload dump	No	xxx	I	E	Write data_in to cache	—
Four-beat write (double-word-aligned)	Reload dump	No	xxx	I	M	Write data_in to cache	—
E⟶I	Snoop write or kill	No	xxx	E	I	State change only (committed)	—
M⟶I	Snoop kill	No	xxx	M	I	State change only (committed)	—
Push M⟶I	Snoop flush	No	xxx	M	I	Conditionally push	Write-with-kill
Push M⟶E	Snoop clean	No	xxx	M	E	Conditionally push	Write-with-kill
tlbie	TLB invalidate	No	xxx	x	x	CRTRY TLBI	—
						Pass TLBI	—
						No action	—
sync	Synchroni-zation	No	xxx	x	x	CRTRY **sync**	—
						Pass **sync**	—
						No action	—

NOTE: Single-beat writes are not snooped in the write queue.

Chapter 4 Exceptions

The OEA portion of the PowerPC architecture defines the mechanism by which PowerPC processors implement exceptions (referred to as interrupts in the architecture specification). Exception conditions may be defined at other levels of the architecture. For example, the UISA defines conditions that may cause floating-point exceptions; the OEA defines the mechanism by which the exception is taken.

The PowerPC exception mechanism allows the processor to change to supervisor state as a result of unusual conditions arising in the execution of instructions and from external signals, bus errors, or various internal conditions. When exceptions occur, information about the state of the processor is saved to certain registers and the processor begins execution at an address (exception vector) predetermined for each exception. Processing of exceptions begins in supervisor mode.

Although multiple exception conditions can map to a single exception vector, often a more specific condition may be determined by examining a register associated with the exception—for example, the DSISR and the floating-point status and control register (FPSCR). Also, software can explicitly enable or disable some exception conditions.

The PowerPC architecture requires that exceptions be taken in program order; therefore, although a particular implementation may recognize exception conditions out of order, they are handled strictly in order with respect to the instruction stream. When an instruction-caused exception is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that have not yet entered the execute state, are required to complete before the exception is taken. For example, if a single instruction encounters multiple exception conditions, those exceptions are taken and handled sequentially. Likewise, exceptions that are asynchronous and precise are recognized when they occur, but are not handled until all instructions currently in the execute stage successfully complete execution and report their results.

To prevent loss of state information, exception handlers must save the information stored in the machine status save/restore registers, SRR0 and SRR1, soon after the exception is taken to prevent this information from being lost due to another exception being taken. Because exceptions can occur while an exception handler routine is executing, multiple exceptions can become nested. It is up to the exception handler to save the necessary state information if control is to return to the excepting program.

In many cases, after the exception handler handles an exception, there is an attempt to execute the instruction that caused the exception. Instruction execution continues until the next exception condition is encountered. Recognizing and handling exception conditions sequentially guarantees that the machine state is recoverable and processing can resume without losing instruction results.

In this book, the following terms are used to describe the stages of exception processing:

Recognition	Exception recognition occurs when the condition that can cause an exception is identified by the processor.
Taken	An exception is said to be taken when control of instruction execution is passed to the exception handler; that is, the context is saved and the instruction at the appropriate vector offset is fetched and the exception handler routine is begun in supervisor mode.
Handling	Exception handling is performed by the software linked to the appropriate vector offset. Exception handling is begun in supervisor mode (referred to as privileged state in the architecture specification).

NOTE: The PowerPC architecture documentation refers to exceptions as interrupts. In this book, the term ‘interrupt’ is reserved to refer to asynchronous exceptions and sometimes to the event that causes the exception. Also, the PowerPC architecture uses the word ‘exception’ to refer to IEEE-defined floating-point exception conditions that may cause a program exception to be taken; see 4.5.7. The occurrence of these IEEE exceptions may not cause an exception to be taken. IEEE-defined exceptions are referred to as IEEE floating-point exceptions or floating-point exceptions.

4.1 PowerPC Gekko Microprocessor Exceptions

As specified by the PowerPC architecture, exceptions can be either precise or imprecise and either synchronous or asynchronous. Asynchronous exceptions are caused by events external to the processor’s execution; synchronous exceptions are caused by instructions.

The types of exceptions are shown in Table 4-1.

NOTE: All exceptions except for the system management interrupt, thermal management, and performance monitor exception are defined, at least to some extent, by the PowerPC architecture.

Table 4-1. PowerPC Gekko Microprocessor Exception Classifications

Synchronous/Asynchronous	Precise/Imprecise	Exception Types
Asynchronous, nonmaskable	Imprecise	Machine check, system reset
Asynchronous, maskable	Precise	External interrupt, decrementer, performance monitor interrupt, thermal management interrupt
Synchronous	Precise	Instruction-caused exceptions

These classifications are discussed in greater detail in Section 4.2, “Exception Recognition and Priorities" on Page 4-4.

For a better understanding of how Gekko implements precise exceptions, see Chapter 6, “Exceptions” of the *PowerPC Microprocessor Family: The Programming Environments* manual. Exceptions implemented in Gekko, and conditions that cause them, are listed in Table 4-2.

Table 4-2. Exceptions and Conditions

Exception Type	Vector Offset (hex)	Causing Conditions
Reserved	00000	—
System reset	00100	Assertion of either $\overline{\text{HRESET}}$ or $\overline{\text{SRESET}}$ or at power-on reset
Machine check	00200	Assertion of $\overline{\text{TEA}}$ during a data bus transaction, assertion of $\overline{\text{MCP}}$, an address, data or L2 double bit error, DMA queue overflow, DMA look-up misses locked cache, or **dcbz_l** cache hit. MSR[ME] must be set.
DSI	00300	As specified in the PowerPC architecture. For TLB misses on load, store, or cache operations, a DSI exception occurs if a page fault occurs.
ISI	00400	As defined by the PowerPC architecture
External interrupt	00500	MSR[EE] = 1 and $\overline{\text{INT}}$ is asserted
Alignment	00600	• A floating-point load/store, **stmw**, **stwcx.**, **lmw**, **lwarx**, **eciwx**, or **ecowx** instruction operand is not word-aligned. • A multiple/string load/store operation is attempted in little-endian mode • An operand of a **dcbz** or **dcbz_l** instruction is on a page that is write-through or cache-inhibited for a virtual mode access. • An attempt to execute a **dcbz** or **dcbz_l** instruction occurs when the cache is disabled.
Program	00700	As defined by the PowerPC architecture
Floating-point unavailable	00800	As defined by the PowerPC architecture
Decrementer	00900	As defined by the PowerPC architecture, when the most-significant bit of the DEC register changes from 0 to 1 and MSR[EE] = 1
Reserved	00A00–00BFF	—
System call	00C00	Execution of the System Call (**sc**) instruction
Trace	00D00	MSR[SE] =1 or a branch instruction is completing and MSR[BE] =1. Gekko differs from the OEA by not taking this exception on an **isync**.
Reserved	00E00	Gekko does not generate an exception to this vector. Other PowerPC processors may use this vector for floating-point assist exceptions.
Reserved	00E10–00EFF	—
Performance monitor	00F00	The limit specified in PMC*n* is met and MMCR0[ENINT] = 1 (Gekko-specific)
Instruction address breakpoint	01300	IABR[0–29] matches EA[0–29] of the next instruction to complete, IABR[TE] matches MSR[IR], and IABR[BE] = 1 (Gekko-specific)
Reserved	01400–016FF	—

Table 4-2. Exceptions and Conditions (Continued)

Exception Type	Vector Offset (hex)	Causing Conditions
Thermal management interrupt	01700	Thermal management is enabled, junction temperature exceeds the threshold specified in THRM1 or THRM2, and MSR[EE] = 1 (Gekko-specific)
Reserved	01800–02FFF	—

4.2 Exception Recognition and Priorities

Exceptions are roughly prioritized by exception class, as follows:

1. Nonmaskable, asynchronous exceptions have priority over all other exceptions—system reset and machine check exceptions (although the machine check exception condition can be disabled so the condition causes the processor to go directly into the checkstop state). These exceptions cannot be delayed and do not wait for completion of any precise exception handling.
2. Synchronous, precise exceptions are caused by instructions and are taken in strict program order.
3. Imprecise exceptions (imprecise mode floating-point enabled exceptions) are caused by instructions and they are delayed until higher priority exceptions are taken. Note that Gekko does not implement an exception of this type.
4. Maskable asynchronous exceptions (external, decrementer, thermal management, system management, performance monitor, and interrupt exceptions) are delayed until higher priority exceptions are taken.

The following list of exception categories describes how Gekko handles exceptions up to the point of signaling the appropriate interrupt to occur. Note that a recoverable state is reached if the completed store queue is empty (drained, not ca..
y instruction that is next in program order and has been signaled to complete has completed. If MSR[RI] = 0, Gekko is in a nonrecoverable state. Also, instruction completion is defined as updating all architectural registers associated with that instruction, and then removing that instruction from the completion buffer.

- Exceptions caused by asynchronous events (interrupts). These exceptions are further distinguished by whether they are maskable and recoverable.
  - — Asynchronous, nonmaskable, nonrecoverable

    System reset for assertion of $\overline{\text{HRESET}}$—Has highest priority and is taken immediately regardless of other pending exceptions or recoverability. (Includes power-on reset)
  - — Asynchronous, maskable, nonrecoverable

    Machine check exception—Has priority over any other pending exception except system reset for assertion of $\overline{\text{HRESET}}$. Taken immediately regardless of recoverability.
  - — Asynchronous, nonmaskable, recoverable

System reset for $\overline{\text{SRESET}}$—Has priority over any other pending exception except system reset for $\overline{\text{HRESET}}$ (or power-on reset), or machine check. Taken immediately when a recoverable state is reached.

— Asynchronous, maskable, recoverable

System management, performance monitor, thermal management, external, and decrementer interrupts—Before handling this type of exception, the next instruction in program order must complete. If that instruction causes another type of exception, that exception is taken and the asynchronous, maskable recoverable exception remains pending, until the instruction completes. Further instruction completion is halted. The asynchronous, maskable recoverable exception is taken when a recoverable state is reached.

- Instruction-related exceptions. These exceptions are further organized into the point in instruction processing in which they generate an exception.

— Instruction fetch

ISI exceptions—Once this type of exception is detected, dispatching stops and the current instruction stream is allowed to drain out of the machine. If completing any of the instructions in this stream causes an exception, that exception is taken and the instruction fetch exception is discarded (but may be encountered again when instruction processing resumes). Otherwise, once all pending instructions have executed and a recoverable state is reached, the ISI exception is taken.

— Instruction dispatch/execution

Program, DSI, alignment, floating-point unavailable, system call, and instruction address breakpoint—This type of exception is determined during dispatch or execution of an instruction. The exception remains pending until all instructions before the exception-causing instruction in program order complete. The exception is then taken without completing the exception-causing instruction. If completing these previous instructions causes an exception, that exception takes priority over the pending instruction dispatch/execution exception, which is then discarded (but may be encountered again when instruction processing resumes).

— Post-instruction execution

Trace—Trace exceptions are generated following execution and completion of an instruction while trace mode is enabled. If executing the instruction produces conditions for another type of exception, that exception is taken and the post-instruction exception is forgotten for that instruction.

NOTE: These exception classifications correspond to how exceptions are prioritized, as described in Table 4-3.

Table 4-3. PowerPC Gekko Exception Priorities

Priority	Exception	Cause
Asynchronous Exceptions (Interrupts)		
0	System reset	Power on reset, assertion of $\overline{\text{HRESET}}$ and $\overline{\text{TRST}}$ (hard reset)
1	Machine check	Any enabled machine check condition (L1 address or data parity error, L2 data double bit error, assertion of $\overline{\text{TEA}}$ or $\overline{\text{MCP}}$)
2	System reset	Assertion of $\overline{\text{SRESET}}$ (soft reset)
3	External interrupt	Assertion of $\overline{\text{INT}}$
4	Performance monitor	Any programmer-specified performance monitor condition
5	Decrementer	Decrementer passes through zero
6	Thermal management	Any programmer-specified thermal management condition
Instruction Fetch Exceptions		
0	ISI	Any ISI exception condition
Instruction Dispatch/Execution Exceptions		
0	Instruction address breakpoint	Any instruction address breakpoint exception condition
1	Program	Occurrence of an illegal instruction, privileged instruction, or trap exception condition. Note that floating-point enabled program exceptions have lower priority.
2	System call	System Call (**sc**) instruction
3	Floating-point unavailable	Any floating-point unavailable exception condition
4	Program	A floating-point enabled exception condition (lowest-priority program exception)
5	DSI	DSI exception due to **eciwx**, **ecowx** with EAR[E] = 0 (DSISR[11]). Lower priority DSI exception conditions are shown below.
6	Alignment	Any alignment exception condition, prioritized as follows: 1 Floating-point access not word-aligned 2 **lmw**, **stmw**, **lwarx**, **stwcx.** not word-aligned 3 **eciwx** or **ecowx** not word-aligned 4 Multiple or string access with MSR[LE] set 5 **dcbz** or **dcbz_l** to write-through or cache-inhibited page or cache is disabled
7	DSI	BAT page protection violation
8	DSI	Any access except cache operations to a segment where SR[T] = 1 (DSISR[5]) or an access crosses from a T = 0 segment to one where T = 1 (DSISR[5])
9	DSI	TLB page protection violation
10	DSI	DABR address match
Post-Instruction Execution Exceptions		

Table 4-3. PowerPC Gekko Exception Priorities (Continued)

Priority	Exception	Cause
11	Trace	MSR[SE] = 1 (or MSR[BE] = 1 for branches)

System reset and machine check exceptions may occur at any time and are not delayed even if an exception is being handled. As a result, state information for an interrupted exception may be lost; therefore, these exceptions are typically nonrecoverable. An exception may not be taken immediately when it is recognized.

4.3 Exception Processing

When an exception is taken, the processor uses SRR0 and SRR1 to save the contents of the MSR for the current context and to identify where instruction execution should resume after the exception is handled.

When an exception occurs, the address saved in SRR0 helps determine where instruction processing should resume when the exception handler returns control to the interrupted process. Depending on the exception, this may be the address in SRR0 or at the next address in the program flow. All instructions in the program flow preceding this one will have completed execution and no subsequent instruction will have begun execution. This may be the address of the instruction that caused the exception or the next one (as in the case of a system call, trace, or trap exception). The SRR0 register is shown in Figure 4-1.

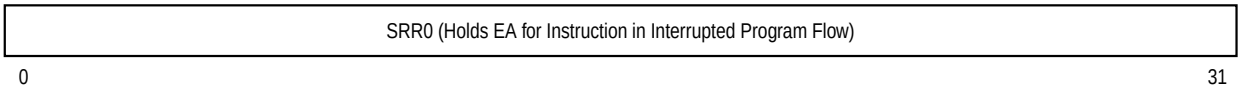


Figure 4-1. Machine Status Save/Restore Register 0 (SRR0)

SRR1 is used to save machine status (selected MSR bits and possibly other status bits as well) on exceptions and to restore those values when an **rfi** instruction is executed. SRR1 is shown in Figure 4-2.

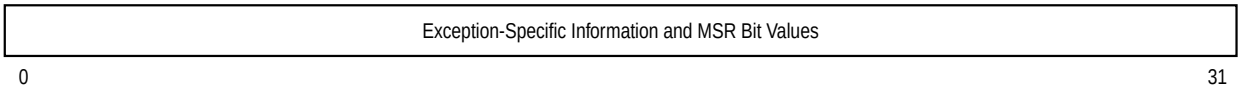


Figure 4-2. Machine Status Save/Restore Register 1 (SRR1)

For most exceptions, bits 2–4 and 10–12 of SRR1 are loaded with exception-specific information and MSR[5–9, 16–31] are placed into the corresponding bit positions of SRR1.

Gekko's MSR is shown in Figure 4-3.

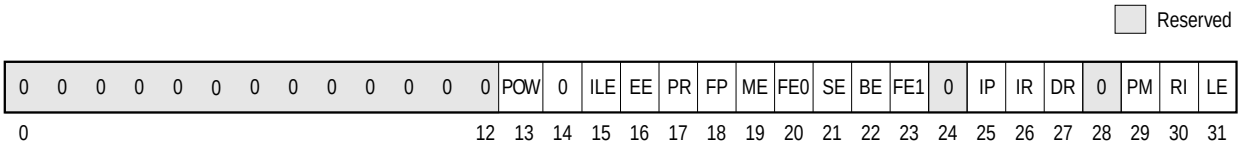


Figure 4-3. Machine State Register (MSR)

The MSR bits are defined in Table 4-4.

Table 4-4. MSR Bit Settings

Bit(s)	Name	Description
0	—	Reserved. Full function.[1]
1–4	—	Reserved. Partial function.[1]
5–9	—	Reserved. Full function.[1]
10–12	—	Reserved. Partial function.[1]
13	POW	Power management enable 0 Power management disabled (normal operation mode). 1 Power management enabled (reduced power mode). Power management functions are implementation-dependent. See Chapter 10, "Power and Thermal Management" in this manual
14	—	Reserved. Implementation-specific
15	ILE	Exception little-endian mode. When an exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception.
16	EE	External interrupt enable 0 The processor delays recognition of external interrupts and decrementer exception conditions. 1 The processor is enabled to take an external interrupt or the decrementer exception.
17	PR	Privilege level 0 The processor can execute both user- and supervisor-level instructions. 1 The processor can only execute user-level instructions.
18	FP	Floating-point available 0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves. 1 The processor can execute floating-point instructions and can take floating-point enabled program exceptions.
19	ME	Machine check enable 0 Machine check exceptions are disabled. 1 Machine check exceptions are enabled.
20	FE0	IEEE floating-point exception mode 0 (see Table 4-5).
21	SE	Single-step trace enable 0 The processor executes instructions normally. 1 The processor generates a single-step trace exception upon the successful execution of every instruction except **rfi**, **isync**, and **sc**. Successful execution means that the instruction caused no other exception.
22	BE	Branch trace enable 0 The processor executes branch instructions normally. 1 The processor generates a branch type trace exception when a branch instruction executes successfully.
23	FE1	IEEE floating-point exception mode 1 (see Table 4-5).
24	—	Reserved. This bit corresponds to the AL bit of the POWER architecture.

Table 4-4. MSR Bit Settings (Continued)

Bit(s)	Name	Description
25	IP	Exception prefix. The setting of this bit specifies whether an exception vector offset is prepended with Fs or 0s. In the following description, *nnnnn* is the offset of the exception. 0 Exceptions are vectored to the physical address 0x000*n_nnnn*. 1 Exceptions are vectored to the physical address 0xFFF*n_nnnn*.
26	IR	Instruction address translation 0 Instruction address translation is disabled. 1 Instruction address translation is enabled. For more information see Chapter 5, "Memory Management" in this manual.
27	DR	Data address translation 0 Data address translation is disabled. 1 Data address translation is enabled. For more information see Chapter 5, "Memory Management" in this manual.
28	—	Reserved. Full function[1]
29	PM	Performance monitor marked mode 0 Process is not a marked process. 1 Process is a marked process. Gekko–specific; defined as reserved by the PowerPC architecture. For more information about the performance monitor, see Section 4.5.13, “Performance Monitor Interrupt (0x00F00)" on Page 4-20.
30	RI	Indicates whether system reset or machine check exception is recoverable. 0 Exception is not recoverable. 1 Exception is recoverable. The RI bit indicates whether from the perspective of the processor, it is safe to continue (that is, processor state data such as that saved to SRR0 is valid), but it does not guarantee that the interrupted process is recoverable.
31	LE	Little-endian mode enable 0 The processor runs in big-endian mode. 1 The processor runs in little-endian mode.

Note: Full function reserved bits are saved in SRR1 when an exception occurs; partial function reserved bits are not saved.

The IEEE floating-point exception mode bits (FE0 and FE1) together define whether floating-point exceptions are handled precisely, imprecisely, or whether they are taken at all. As shown in Table 4-5, if either FE0 or FE1 are set, Gekko treats exceptions as precise. MSR bits are guaranteed to be written to SRR1 when the first instruction of the exception handler is encountered. For further details, see Chapter 6, “Exceptions” of the *PowerPC Microprocessor Family: The Programming Environments* manual.

Table 4-5. IEEE Floating-Point Exception Mode Bits

FE0	FE1	Mode
0	0	Floating-point exceptions disabled
0	1	Imprecise nonrecoverable. For this setting, Gekko operates in floating-point precise mode.
1	0	Imprecise recoverable. For this setting, Gekko operates in floating-point precise mode.

Table 4-5. IEEE Floating-Point Exception Mode Bits (Continued)

FE0	FE1	Mode
1	1	Floating-point precise mode

4.3.1 Enabling and Disabling Exceptions

When a condition exists that may cause an exception to be generated, it must be determined whether the exception is enabled for that condition.

- IEEE floating-point enabled exceptions (a type of program exception) are ignored when both MSR[FE0] and MSR[FE1] are cleared. If either bit is set, all IEEE enabled floating-point exceptions are taken and cause a program exception.
- Asynchronous, maskable exceptions (such as the external and decrementer interrupts) are enabled by setting MSR[EE]. When MSR[EE] = 0, recognition of these exception conditions is delayed. MSR[EE] is cleared automatically when an exception is taken to delay recognition of conditions causing those exceptions.
- A machine check exception can occur only if the machine check enable bit, MSR[ME], is set. If MSR[ME] is cleared, the processor goes directly into checkstop state when a machine check exception condition occurs. Individual machine check exceptions can be enabled and disabled through bits in the HID0 register, which is described in Table 4-9.
- System reset exceptions cannot be masked.

4.3.2 Steps for Exception Processing

After it is determined that the exception can be taken (by confirming that any instruction-caused exceptions occurring earlier in the instruction stream have been handled, and by confirming that the exception is enabled for the exception condition), the processor does the following:

1. SRR0 is loaded with an instruction address that depends on the type of exception. See the individual exception description for details about how this register is used for specific exceptions.
2. SRR1[1–4, 10–15] are loaded with information specific to the exception type.
3. SRR1[5–9, 16–31] are loaded with a copy of the corresponding MSR bits. Depending on the implementation, reserved bits may not be copied.
4. The MSR is set as described in Table 4-4. The new values take effect as the first instruction of the exception-handler routine is fetched.

   Note that MSR[IR] and MSR[DR] are cleared for all exception types; therefore, address translation is disabled for both instruction fetches and data accesses beginning with the first instruction of the exception-handler routine.
5. Instruction fetch and execution resumes, using the new MSR value, at a location specific to the exception type. The location is determined by adding the exception's vector (see Table 4-2 on Page 4-3) to the base address determined by MSR[IP]. If IP is cleared, exceptions are vectored to the physical address 0x000*n_nnnn*. If IP is set, exceptions are vectored to the physical address 0xFFF*n_nnnn*. For a machine check exception that occurs when MSR[ME] = 0 (machine check exceptions are disabled), the checkstop state is entered (the machine stops executing instructions). See .”

4.3.3 Setting MSR[RI]

An operating system may handle MSR[RI] as follows:

- In the machine check and system reset exceptions—If MSR[RI] is cleared, the exception is not recoverable. If it is set, the exception is recoverable with respect to the processor.
- In each exception handler—When enough state information has been saved that a machine check or system reset exception can reconstruct the previous state, set MSR[RI].
- In each exception handler—Clear MSR[RI], set SRR0 and SRR1 appropriately, and then execute **rfi**.
- Note that the RI bit being set indicates that, with respect to the processor, enough processor state data remains valid for the processor to continue, but it does not guarantee that the interrupted process can resume.

4.3.4 Returning from an Exception Handler

The Return from Interrupt (**rfi**) instruction performs context synchronization by allowing previously-issued instructions to complete before returning to the interrupted process. In general, execution of the **rfi** instruction ensures the following:

- All previous instructions have completed to a point where they can no longer cause an exception. If a previous instruction causes a direct-store interface error exception, the results must be determined before this instruction is executed.
- Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
- The **rfi** instruction copies SRR1 bits back into the MSR.
- Instructions fetched after this instruction execute in the context established by this instruction.
- Program execution resumes at the instruction indicated by SRR0

For a complete description of context synchronization, refer to Chapter 6, “Exceptions” of the *PowerPC Microprocessor Family: The Programming Environments* manual.

4.4 Process Switching

The following instructions are useful for restoring proper context during process switching:

- The **sync** instruction orders the effects of instruction execution. All instructions previously initiated appear to have completed before the **sync** instruction completes, and no subsequent instructions appear to be initiated until the **sync** instruction completes. For an example showing use of **sync**, see Chapter 2, “PowerPC Register Set” of the *PowerPC Microprocessor Family: The Programming Environments* manual.
- The **isync** instruction waits for all previous instructions to complete and then discards any fetched instructions, causing subsequent instructions to be fetched (or refetched) from memory and to execute in the context (privilege, translation, and protection) established by the previous instructions.
- The **stwcx.** instruction clears any outstanding reservations, ensuring that an **lwarx** instruction in an old process is not paired with an **stwcx.** instruction in a new one.

The operating system should set MSR[RI] as described in 4.3.3.”

4.5 Exception Definitions

Table 4-6 shows all the types of exceptions that can occur with Gekko and MSR settings when the processor goes into supervisor mode due to an exception. Depending on the exception, certain of these bits are stored in SRR1 when an exception is taken.

Table 4-6. MSR Setting Due to Exception

Exception Type	MSR Bit[1]															
	POW	ILE	EE	PR	FP	ME	FE0	SE	BE	FE1	IP	IR	DR	PM	RI	LE
System reset	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Machine check	0	—	0	0	0	0	0	0	0	0	—	0	0	0	0	ILE
DSI	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
ISI	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
External interrupt	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Alignment	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Program	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Floating-point unavailable	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Decrementer interrupt	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
System call	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Trace exception	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Performance monitor	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Thermal management	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE

Note:

1. 0 Bit is cleared.
   ILEBit is copied from the MSR[ILE].
   — Bit is not altered
   Reserved bits are read as if written as 0.

The setting of the exception prefix bit (IP) determines how exceptions are vectored. If the bit is cleared, exceptions are vectored to the physical address 0x000*n_nnnn* (where *nnnnn* is the vector offset); if IP is set, exceptions are vectored to physical address 0xFFF*n_nnnn*. Table 4-2 on Page 4-3 shows the exception vector offset of the first instruction of the exception handler routine for each exception type.

4.5.1 System Reset Exception (0x00100)

Gekko implements the system reset exception as defined in the PowerPC architecture (OEA). The system reset exception is a nonmaskable, asynchronous exception signaled to the processor through the assertion of system-defined signals. In Gekko, the exception is signaled by the assertion of either the soft reset ($\overline{\text{SRESET}}$) or hard reset ($\overline{\text{HRESET}}$) inputs, described more fully in Chapter 7, "Signal Descriptions" in this manual

Gekko implements HID0[NHR], which helps software distinguish a hard reset from a soft reset. Because this bit is cleared by a hard reset, but not by a soft reset, software can set this bit after a hard reset and tell whether a subsequent reset is a hard or soft reset by examining whether this bit is still set.

The first bus operation following the negation of $\overline{\text{HRESET}}$ or the assertion of SRESET will be a single-beat instruction fetch (caching will be inhibited) to x00100.

Table 4-7 lists register settings when a system reset exception is taken.

Table 4-7. System Reset Exception—Register Settings

Register	Setting Description			
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.			
SRR1	0 Loaded with equivalent MSR bits 1–4 Cleared 5–9 Loaded with equivalent MSR bits 10–15 Cleared 16–31 Loaded with equivalent MSR bits Note that if the processor state is corrupted to the extent that execution cannot resume reliably, MSR[RI] (SRR1[30]) is cleared.			
MSR	POW 0 ILE — EE 0 PR 0	FP 0 ME — FE0 0 SE 0	BE 0 FE1 0 IP — IR 0	DR 0 PM 0 RI 0 LE Set to value of ILE

4.5.1.1 Soft Reset

If $\overline{\text{SRESET}}$ is asserted, the processor is first put in a recoverable state. To do this, Gekko allows any instruction at the point of completion to either complete or take an exception, blocks completion of any following instructions, and allows the completion queue to drain. The state before the exception occurred is then saved as specified in the PowerPC architecture and instruction fetching begins at the system reset interrupt vector offset, 0x00100. The vector address on a soft reset depends on the setting of MSR[IP] (either 0x0000_0100 or 0xFFF0_0100). Soft resets are third in priority, after hard reset and machine check. This exception is recoverable provided attaining a recoverable state does not generate a machine check.

$\overline{\text{SRESET}}$ is an effectively edge-sensitive signal that can be asserted and deasserted asynchronously, provided the minimum pulse width specified in the hardware specifications is met. Asserting $\overline{\text{SRESET}}$ causes Gekko to take a system reset exception. This exception modifies the MSR, SRR0, and SRR1, as described in the *PowerPC Microprocessor Family: The Programming Environments* manual. Unlike hard reset, soft reset does not directly affect the states of output signals. Attempts to use $\overline{\text{SRESET}}$ during a hard reset sequence or while the JTAG logic is non-idle cause unpredictable results (see Section 7.2.9.5.2, “Soft Reset (SRESET)—Input" on Page 7-17 for more information on soft reset).

$\overline{\text{SRESET}}$ can be asserted during $\overline{\text{HRESET}}$ assertion (see Figure 4-4). In all three cases shown in Figure 4-4, the $\overline{\text{SRESET}}$ assertion and deassertion have no effect on the operation or state of the machine. $\overline{\text{SRESET}}$ asserted coincident to, or after the assertion of, $\overline{\text{HRESET}}$ will also have no effect on the operation or state of the machine.

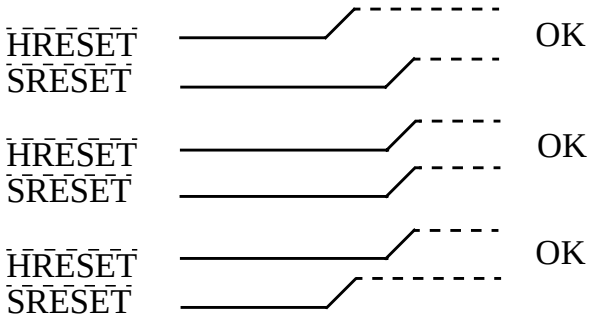


Figure 4-4. SRESET Asserted During HRESET

4.5.1.2 Hard Reset

A hard reset is initiated by asserting HRESET. Hard reset is used primarily for power-on reset (POR) (in which case TRST must also be asserted), but it can also be used to restart a running processor. The HRESET signal must be asserted during power up and must remain asserted for a period that allows the PLL to achieve lock and the internal logic to be reset. This period is specified in the hardware specifications. Gekko tri-states all IO drivers within five clocks of HRESET assertion. Gekko's internal state after the hard reset interval is defined in Table 4-8. If HRESET is asserted for less than this amount of time, the results are not predictable. If HRESET is asserted during normal operation, all operations cease, and the machine state is lost (see Section 7.2.9.5.1, "Hard Reset (HRESET)—Input" on Page 7-17 for more information on a hard reset).

The hard reset exception is a nonrecoverable, nonmaskable asynchronous exception. When HRESET is asserted or at power-on reset (POR), Gekko immediately branches to 0xFFF0_0100 without attempting to reach a recoverable state. A hard reset has the highest priority of any exception. It is always nonrecoverable. Table 4-8 shows the state of the machine just before it fetches the first instruction of the system reset handler after a hard reset. In Table 4-8, the term "Unknown" means that the content may have been disordered. These facilities must be properly initialized before use. The FPRs, BATs, and TLBs may have been disordered. To initialize the BATs, first set them all to zero, then to the correct values before any address translation occurs.

Table 4-8. Settings Caused by Hard Reset

Register	Setting	Register	Setting
GPRs	Unknown	PVR	see the *PowerPC* Gekko *Microprocessor Data Sheet*
FPRs	Unknown	HID0	00000000
FPSCR	00000000	HID1	00000000
CR	All 0s	HID2	00000000
SRs	Unknown	GQRn	00000000
MSR	00000040 (only IP set)	WPAR	00000000
XER	00000000	IABR	All 0s (break point disabled)

Table 4-8. Settings Caused by Hard Reset (Continued)

Register	Setting	Register	Setting
TBU	00000000	DSISR	00000000
TBL	00000000	DAR	00000000
LR	00000000	DEC	FFFFFFFF
CTR	00000000	DMAU	00000000
SDR1	00000000	DMAL	00000000
SRR0	00000000	TLBs	Unknown
SRR1	00000000	Reservation Address	Unknown (reservation flag -cleared)
SPRGs	00000000	BATs	Unknown
Tag directory, Icache, and Dcache	All entries are marked invalid, all LRU bits are set to 0, and caches are disabled.	Cache, Icache, and Dcache	All blocks are unchanged from before $\overline{\text{HRESET}}$.
DABR	Breakpoint is disabled. Address is unknown.		
L2CR	00000000		
MMCR*n*	00000000		
THRM*n*	00000000		
UMMCR*n*	00000000		
UPMC*n*	00000000		
USIA	00000000		
XER	00000000		
PMC*n*	Unknown		
ICTC	00000000		

The following is also true after a hard reset operation:

- External checkstops are enabled.
- The on-chip test interface has given control of the I/Os to the rest of the chip for functional use.
- Since the reset exception has data and instruction translation disabled (MSR[DR] and MSR[IR] both cleared), the chip operates in direct address translation mode (referred to as the real addressing mode in the architecture specification).
- Time from HRESET deassertion until Gekko asserts the first $\overline{\text{TS}}$ (bus parked on Gekko) or $\overline{\text{BG}}$ is 8 to 12 bus clocks (SYSCLK).

4.5.2 Machine Check Exception (0x00200)

Gekko implements the machine check exception as defined in the PowerPC architecture (OEA). It conditionally initiates a machine check exception after an address or data parity error occurred on the bus or in either the L1 or L2 cache, after receiving a qualified transfer error acknowledge ($\overline{TEA}$) indication on Gekko bus, after DMA look-up missed the locked cache, after a **dcbz_l** hit in the normal cache, or after the machine check interrupt ($\overline{MCP}$) signal had been asserted. As defined in the OEA, the exception is not taken if MSR[ME] is cleared, in which case the processor enters checkstop state.

Certain machine check conditions can be enabled and disabled using HID0 bits, as described in Table 4-9.

Table 4-9. HID0 Machine Check Enable Bits

Bit	Name	Function
0	EMCP	Enable $\overline{MCP}$. The primary purpose of this bit is to mask out further machine check exceptions caused by assertion of $\overline{MCP}$, similar to how MSR[EE] can mask external interrupts. 0 Masks $\overline{MCP}$. Asserting $\overline{MCP}$ does not generate a machine check exception or a checkstop. 1 Asserting $\overline{MCP}$ causes a checkstop if MSR[ME] = 0 or a machine check exception if MSR[ME] = 1.
1	DBP	Enable/disable 60x bus address and data parity generation. 0 If address or data parity is not used by the system and the respective parity checking is disabled (HID0[EBA] or HID0[EBD] = 0), input receivers for those signals are disabled, do not require pull-up resistors, and therefore should be left unconnected. If all parity generation is disabled, all parity checking should also be disabled and parity signals need not be connected. 1 Parity generation is enabled.
2	EBA	Enable/disable 60x bus address parity checking. 0 Prevents address parity checking. 1 Allows a address parity error to cause a checkstop if MSR[ME] = 0 or a machine check exception if MSR[ME] = 1. EBA and EBD allow the processor to operate with memory subsystems that do not generate parity.
3	EBD	Enable 60x bus data parity checking 0 Parity checking is disabled. 1 Allows a data parity error to cause a checkstop if MSR[ME] = 0 or a machine check exception if MSR[ME] = 1. EBA and EBD allow the processor to operate with memory subsystems that do not generate parity.
15	NHR	Not hard reset (software use only) 0 A hard reset occurred if software had previously set this bit 1 A hard reset has not occurred.

A $\overline{TEA}$ indication on the bus can result from any load or store operation initiated by the processor. In general, $\overline{TEA}$ is expected to be used by a memory controller to indicate that a memory parity error or an uncorrectable memory ECC error has occurred. Note that the resulting machine check exception is imprecise and unordered with respect to the instruction that originated the bus operation.

If MSR[ME] and the appropriate HID0 bits are set, the exception is recognized and handled; otherwise, the processor generates an internal checkstop condition. When the exception is recognized, all incomplete stores are discarded. The bus protocol operates normally.

A machine check exception may result from referencing a nonexistent physical address, either directly (with MSR[DR] = 0) or through an invalid translation. If a **dcbz** instruction introduces a block into the cache associated with a nonexistent physical address, a machine check exception can be delayed until an attempt is made to store that block to main memory. Not all PowerPC processors provide the same level of error checking. Checkstop sources are implementation-dependent.

Machine check exceptions are enabled when MSR[ME] = 1; this is described in the next section.. If MSR[ME] = 0 and a machine check occurs, the processor enters the checkstop state.

Checkstop state is described in Section 4.5.2.2, “Checkstop State (MSR[ME] = 0)" on Page 4-17.

4.5.2.1 Machine Check Exception Enabled (MSR[ME] = 1)

Machine check exceptions are enabled when MSR[ME] = 1. When a machine check exception is taken, registers are updated as shown in Table 4-10.

Table 4-10. Machine Check Exception—Register Settings

Register	Setting Description			
SRR0	On a best-effort basis Gekko can set this to an EA of some instruction that was executing or about to be executing when the machine check condition occurred.			
SRR1	0–9 Cleared 10 Set when a DMA or locked cache error happens. 11 Set when an L2 data cache double bit error is detected, otherwise zero 12 Set when $\overline{\text{MCP}}$ signal is asserted, otherwise zero 13 Set when $\overline{\text{TEA}}$ signal is asserted, otherwise zero 14 Set when a data bus parity error is detected, otherwise zero 15 Set when an address bus parity error is detected, otherwise zero 16–31 MSR[16–31]			
	POW 0 ILE — EE 0 PR 0	FP 0 ME 0 FE0 0 SE 0	BE 0 FE1 0 IP — IR 0	DR 0 PM 0 RI 0 LE Set to value of ILE

To handle another machine check exception, the exception handler should set MSR[ME] as soon as it is practical after a machine check exception is taken. Otherwise, subsequent machine check exceptions cause the processor to enter the checkstop state.

The machine check exception is usually unrecoverable in the sense that execution cannot resume in the context that existed before the exception. If the condition that caused the machine check does not otherwise prevent continued execution, MSR[ME] is set to allow the processor to continue execution at the machine check exception vector address. Typically, earlier processes cannot resume; however, operating systems can use the machine check exception handler to try to identify and log the cause of the machine check condition.

When a machine check exception is taken, instruction fetching resumes at offset 0x00200 from the physical base address indicated by MSR[IP].

4.5.2.2 Checkstop State (MSR[ME] = 0)

If MSR[ME] = 0 and a machine check occurs, the processor enters the checkstop state. In addition, the assertion of $\overline{\text{CKSTP\_IN}}$ to Gekko causes checkstop.

When a processor is in checkstop state, instruction processing is suspended and generally cannot resume without the processor being reset. The contents of all latches are frozen within two cycles upon entering checkstop state.

4.5.3 DSI Exception (0x00300)

A DSI exception occurs when no higher priority exception exists and an error condition related to a data memory access occurs. The DSI exception is implemented as it is defined in the PowerPC

architecture (OEA). In case of a TLB miss for a load, store, or cache operation, a DSI exception is taken if the resulting hardware table search causes a page fault.

On Gekko, a DSI exception is taken when a load or store is attempted to a direct-store segment (SR[T] = 1). In Gekko, a floating-point load or store to a direct-store segment causes a DSI exception rather than an alignment exception, as specified by the PowerPC architecture.

Gekko also implements the data address breakpoint facility, which is defined as optional in the PowerPC architecture and is supported by the optional data address breakpoint register (DABR). Although the architecture does not strictly prescribe how this facility must be implemented, Gekko follows the recommendations provided by the architecture and described in the Chapter 2, "Programming Model" in this manual and Chapter 6, “Exceptions” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

4.5.4 ISI Exception (0x00400)

An ISI exception occurs when no higher priority exception exists and an attempt to fetch the next instruction fails. This exception is implemented as it is defined by the PowerPC architecture (OEA), and is taken for the following conditions:

- The effective address cannot be translated.
- The fetch access is to a no-execute segment (SR[N] = 1).
- The fetch access is to guarded storage and MSR[IR] = 1.
- The fetch access is to a segment for which SR[T] is set.
- The fetch access violates memory protection.

When an ISI exception is taken, instruction fetching resumes at offset 0x00400 from the physical base address indicated by MSR[IP].

4.5.5 External Interrupt Exception (0x00500)

An external interrupt is signaled to the processor by the assertion of the external interrupt signal ($\overline{\text{INT}}$). The $\overline{\text{INT}}$ signal is expected to remain asserted until Gekko takes the external interrupt exception. If $\overline{\text{INT}}$ is negated early, recognition of the interrupt request is not guaranteed. After Gekko begins execution of the external interrupt handler, the system can safely negate the $\overline{\text{INT}}$. When Gekko detects assertion of $\overline{\text{INT}}$, it stops dispatching and waits for all pending instructions to complete. This allows any instructions in progress that need to take an exception to do so before the external interrupt is taken. After all instructions have vacated the completion buffer, Gekko takes the external interrupt exception as defined in the PowerPC architecture (OEA).

An external interrupt may be delayed by other higher priority exceptions or if MSR[EE] is cleared when the exception occurs. Register settings for this exception are described in Chapter 6, “Exceptions” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

When an external interrupt exception is taken, instruction fetching resumes at offset 0x00500 from the physical base address indicated by MSR[IP].

4.5.6 Alignment Exception (0x00600)

Gekko implements the alignment exception as defined by the PowerPC architecture (OEA). An alignment exception is initiated when any of the following occurs:

- The operand of a floating-point load or store is not word-aligned.
- The operand of **lmw**, **stmw**, **lwarx**, or **stwcx.** is not word-aligned.
- The operand of **dcbz** or **dcbz_l** is in a page which is write-through or cache-inhibited.
- An attempt is made to execute **dcbz** or **dcbz_l** when the data cache is disabled.
- An **eciwx** or **ecowx** is not word-aligned.
- A multiple or string access is attempted with MSR[LE] set.

NOTE: In Gekko, the paired-single quantization load or store will generate an alignment exception when the corresponding GQRn[LD_TYPE] or GQRn[ST_TYPE] are 0 and will not generate an alignment exception when the corresponding GQRn[LD_TYPE] or GQRn[ST_TYPE] are 4, 5, 6 or 7. Also, a floating-point load or store to a direct-store segment causes a DSI exception rather than an alignment exception, as specified by the PowerPC architecture. For more information, see Section 4.5.3, “DSI Exception (0x00300)" on Page 4-17.

4.5.7 Program Exception (0x00700)

Gekko implements the program exception as it is defined by the PowerPC architecture (OEA). A program exception occurs when no higher priority exception exists and one or more of the exception conditions defined in the OEA occur.

Gekko invokes the system illegal instruction program exception when it detects any instruction from the illegal instruction class. Gekko fully decodes the SPR field of the instruction. If an undefined SPR is specified, a program exception is taken.

The UISA defines **mtspr** and **mfspr** with the record bit (Rc) set as causing a program exception or giving a boundedly-undefined result. In Gekko, the appropriate condition register (CR) should be treated as undefined. Likewise, the PowerPC architecture states that the Floating Compared Unordered (**fcmpu**) or Floating Compared Ordered (**fcmpo**) instruction with the record bit set can either cause a program exception or provide a boundedly-undefined result. In Gekko, an the BF field in an instruction encoding for these cases is considered undefined.

Gekko does not support either of the two floating-point imprecise modes supported by the PowerPC architecture. Unless exceptions are disabled (MSR[FE0] = MSR[FE1] = 0), all floating-point exceptions are treated as precise.

When a program exception is taken, instruction fetching resumes at offset 0x00700 from the physical base address indicated by MSR[IP]. Chapter 6, “Exceptions” in the *PowerPC Microprocessor Family: The Programming Environments* manual describes register settings for this exception.

4.5.8 Floating-Point Unavailable Exception (0x00800)

The floating-point unavailable exception is implemented as defined in the PowerPC architecture. A floating-point unavailable exception occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point load, store, or move instructions), and the floating-point available bit in the MSR is disabled, (MSR[FP] = 0). Register settings for this exception are described in Chapter 6, “Exceptions” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

When a floating-point unavailable exception is taken, instruction fetching resumes at offset 0x00800 from the physical base address indicated by MSR[IP].

4.5.9 Decrementer Exception (0x00900)

The decrementer exception is implemented in Gekko as it is defined by the PowerPC architecture. The decrementer exception occurs when no higher priority exception exists, a decrementer exception condition occurs (for example, the decrementer register has completed decrementing), and MSR[EE] = 1. In Gekko, the decrementer register is decremented at one fourth the bus clock rate. Register settings for this exception are described in Chapter 6, “Exceptions” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

When a decrementer exception is taken, instruction fetching resumes at offset 0x00900 from the physical base address indicated by MSR[IP].

4.5.10 System Call Exception (0x00C00)

A system call exception occurs when a System Call (**sc**) instruction is executed. In Gekko, the system call exception is implemented as it is defined in the PowerPC architecture. Register settings for this exception are described in Chapter 6, “Exceptions” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

When a system call exception is taken, instruction fetching resumes at offset 0x00C00 from the physical base address indicated by MSR[IP].

4.5.11 Trace Exception (0x00D00)

The trace exception is taken if MSR[SE] = 1 or if MSR[BE] = 1 and the currently completing instruction is a branch. Each instruction considered during trace mode completes before a trace exception is taken.

Implementation Note—Gekko processor diverges from the PowerPC architecture in that it does not take trace exceptions on the **isync** instruction.

When a trace exception is taken, instruction fetching resumes as offset 0x00D00 from the base address indicated by MSR[IP].

4.5.12 Floating-Point Assist Exception (0x00E00)

The optional floating-point assist exception defined by the PowerPC architecture is not implemented in Gekko.

4.5.13 Performance Monitor Interrupt (0x00F00)

Gekko microprocessor provides a performance monitor facility to monitor and count predefined events such as processor clocks, misses in either the instruction cache or the data cache, instructions dispatched to a particular execution unit, mispredicted branches, and other occurrences. The count of such events can be used to trigger the performance monitor exception. The performance monitor facility is not defined by the PowerPC architecture.

The performance monitor can be used for the following:

- To increase system performance with efficient software, especially in a multiprocessing system. Memory hierarchy behavior must be monitored and studied to develop algorithms that schedule tasks (and perhaps partition them) and that structure and distribute data optimally.
- To help system developers bring up and debug their systems.

The performance monitor uses the following SPRs:

- The performance monitor counter registers (PMC1–PMC4) are used to record the number of times a certain event has occurred. UPMC1–UPMC4 provide user-level read access to these registers.
- The monitor mode control registers (MMCR0–MMCR1) are used to enable various performance monitor interrupt functions. UMMCR0–UMMCR1 provide user-level read access to these registers.

- The sampled instruction address register (SIA) contains the effective address of an instruction executing at or around the time that the processor signals the performance monitor interrupt condition. The USIA register provides user-level read access to the SIA.

Table 4-11 lists register settings when a performance monitor interrupt exception is taken.

Table 4-11. Performance Monitor Interrupt Exception—Register Settings

Register	Setting Description			
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.			
SRR1	0 Loaded with equivalent MSR bits 1–4 Cleared 5–9 Loaded with equivalent MSR bits 10–15 Cleared 16–31 Loaded with equivalent MSR bits			
MSR	POW 0 ILE — EE 0 PR 0	FP 0 ME — FE0 0 SE 0	BE 0 FE1 0 IP — IR 0	DR 0 PM 0 RI 0 LE Set to value of ILE

As with other PowerPC exceptions, the performance monitor interrupt follows the normal PowerPC exception model with a defined exception vector offset (0x00F00). The priority of the performance monitor interrupt lies between the external interrupt and the decrementer interrupt (see Table 4-3). The contents of the SIA are described in 2.1.2.4, “Hardware Implementation-Dependent Register 2.” The performance monitor is described in Chapter 11, "Performance Monitor" in this manual.

4.5.14 Instruction Address Breakpoint Exception (0x01300)

An instruction address breakpoint interrupt occurs when the following conditions are met:

- The instruction breakpoint address IABR[0–29] matches EA[0–29] of the next instruction to complete in program order. The instruction that triggers the instruction address breakpoint exception is not executed before the exception handler is invoked.
- The translation enable bit (IABR[TE]) matches MSR[IR].
- The breakpoint enable bit (IABR[BE]) is set. The address match is also reported to the JTAG/COP block, which may subsequently generate a soft or hard reset. The instruction tagged with the match does not complete before the breakpoint exception is taken.

Table 4-12 lists register settings when an instruction address breakpoint exception is taken.

Table 4-12. Instruction Address Breakpoint Exception—Register Settings

Register	Setting Description
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.
SRR1	0 Loaded with equivalent MSR bits 1–4 Cleared 5–9 Loaded with equivalent MSR bits 10–15 Cleared 16–31 Loaded with equivalent MSR bits

Table 4-12. Instruction Address Breakpoint Exception—Register Settings (Contin-

MSR	POW 0 ILE — EE 0 PR 0	FP 0 ME — FE0 0 SE 0	BE 0 FE1 0 IP — IR 0	DR 0 PM 0 RI 0 LE Set to value of ILE

Gekko requires that an **mtspr** to the IABR be followed by a context-synchronizing instruction. Gekko cannot generate a breakpoint response for that context-synchronizing instruction if the breakpoint is enabled by the **mtspr(IABR)** immediately preceding it. Gekko also cannot block a breakpoint response on the context-synchronizing instruction if the breakpoint was disabled by the **mtspr(IABR)** instruction immediately preceding it. The format of the IABR register is shown in 2.1.2.1."

When an instruction address breakpoint exception is taken, instruction fetching resumes as offset 0x01300 from the base address indicated by MSR[IP].

4.5.15 Thermal Management Interrupt Exception (0x01700)

A thermal management interrupt is generated when the junction temperature crosses a threshold programmed in either THRM1 or THRM2. The exception is enabled by the TIE bit of either THRM1 or THRM2, and can be masked by setting MSR[EE].

Table 4-13 lists register settings when a thermal management interrupt exception is taken.

Table 4-13. Thermal Management Interrupt Exception—Register Settings

Register	Setting Description			
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.			
SRR1	0 Loaded with equivalent MSR bits 1–4 Cleared 5–9 Loaded with equivalent MSR bits 10–15 Cleared 16–31 Loaded with equivalent MSR bits			
MSR	POW 0 ILE — EE 0 PR 0	FP 0 ME — FE0 0 SE 0	BE 0 FE1 0 IP — IR 0	DR 0 PM 0 RI 0 LE Set to value of ILE

The thermal management interrupt is similar to the system management and external interrupts. Gekko requires the next instruction in program order to complete or take an exception, blocks completion of any following instructions, and allows the completed store queue to drain. Any exceptions encountered in this process are taken first and the thermal management interrupt exception is delayed until a recoverable halt is achieved, at which point Gekko saves the machine state, as shown in Table 4-13. When a thermal management interrupt exception is taken, instruction fetching resumes as offset 0x01700 from the base address indicated by MSR[IP].

Chapter 10, "Power and Thermal Management" in this manual gives the details about thermal management.

Chapter 5 Memory Management

This chapter describes Gekko microprocessor's implementation of the memory management unit (MMU) specifications provided by the operating environment architecture (OEA) for PowerPC processors. The primary function of the MMU in a PowerPC processor is the translation of logical (effective) addresses to physical addresses (referred to as real addresses in the architecture specification) for memory accesses and I/O accesses (I/O accesses are assumed to be memory-mapped). In addition, the MMU provides access protection on a segment, block, or page basis. This chapter describes the specific hardware used to implement the MMU model of the OEA in Gekko. Refer to Chapter 7, "Memory Management," in the *PowerPC Microprocessor Family: The Programming Environments* manual for a complete description of the conceptual model. Note that Gekko does not implement the optional direct-store facility and it is not likely to be supported in future devices.

Two general types of memory accesses generated by PowerPC processors require address translation—instruction accesses and data accesses generated by load and store instructions. Generally, the address translation mechanism is defined in terms of the segment descriptors and page tables PowerPC processors use to locate the effective-to-physical address mapping for memory accesses. The segment information translates the effective address to an interim virtual address, and the page table information translates the interim virtual address to a physical address.

The segment descriptors, used to generate the interim virtual addresses, are stored as on-chip segment registers on 32-bit implementations (such as Gekko). In addition, two translation lookaside buffers (TLBs) are implemented on Gekko to keep recently-used page address translations on-chip. Although the PowerPC OEA describes one MMU (conceptually), Gekko hardware maintains separate TLBs and table search resources for instruction and data accesses that can be performed independently (and simultaneously). Therefore, Gekko is described as having two MMUs, one for instruction accesses (IMMU) and one for data accesses (DMMU).

The block address translation (BAT) mechanism is a software-controlled array that stores the available block address translations on-chip. BAT array entries are implemented as pairs of BAT registers that are accessible as supervisor special-purpose registers (SPRs). There are separate instruction and data BAT mechanisms, and in Gekko, they reside in the instruction and data MMUs, respectively.

The MMUs, together with the exception processing mechanism, provide the necessary support for the operating system to implement a paged virtual memory environment and for enforcing protection of designated memory areas.

Exception processing is described in Chapter 4, "Exceptions" specifically, Section 4.3 on Page 4-7 describes the MSR, which controls some of the critical functionality of the MMUs.

5.1 MMU Overview

Gekko implements the memory management specification of the PowerPC OEA for 32-bit implementations. Thus, it provides 4 Gbytes of effective address space accessible to supervisor and user programs, with a 4-Kbyte page size and 256-Mbyte segment size. In addition, the MMUs of 32-bit PowerPC processors use an interim virtual address (52 bits) and hashed page tables in the generation of 32-bit physical addresses. PowerPC processors also have a BAT mechanism for mapping large blocks of memory. Block sizes range from 128 Kbyte to 256 Mbyte and are software-programmable.

Basic features of Gekko MMU implementation defined by the OEA are as follows:

- Support for real addressing mode—Effective-to-physical address translation can be disabled separately for data and instruction accesses.
- Block address translation—Each of the BAT array entries (four IBAT entries and four DBAT entries) provides a mechanism for translating blocks as large as 256 Mbytes from the 32-bit effective address space into the physical memory space. This can be used for translating large address ranges whose mappings do not change frequently.
- Segmented address translation—The 32-bit effective address is extended to a 52-bit virtual address by substituting 24 bits of upper address bits from the segment register, for the 4 upper bits of the EA, which are used as an index into the segment register file. This 52-bit virtual address space is divided into 4-Kbyte pages, each of which can be mapped to a physical page.

Gekko also provides the following features that are not required by the PowerPC architecture:

- Separate translation lookaside buffers (TLBs)—The 128-entry, two-way set-associative ITLBs and DTLBs keep recently-used page address translations on-chip.
- Table search operations performed in hardware—The 52-bit virtual address is formed and the MMU attempts to fetch the PTE, which contains the physical address, from the appropriate TLB on-chip. If the translation is not found in a TLB (that is, a TLB miss occurs), the hardware performs a table search operation (using a hashing function) to search for the PTE.
- TLB invalidation—Gekko implements the optional TLB Invalidate Entry (**tlbie**) and TLB Synchronize (**tlbsync**) instructions, which can be used to invalidate TLB entries. For more information on the **tlbie** and **tlbsync** instructions, see 5.4.3.2.”

Table 5-1 summarizes Gekko MMU features, including those defined by the PowerPC architecture (OEA) for 32-bit processors and those specific to Gekko.

Table 5-1. MMU Feature Summary

Feature Category	Architecturally Defined/ Gekko-Specific	Feature
Address ranges	Architecturally defined	2^{32} bytes of effective address
		2^{52} bytes of virtual address
		2^{32} bytes of physical address
Page size	Architecturally defined	4 Kbytes
Segment size	Architecturally defined	256 Mbytes
Block address translation	Architecturally defined	Range of 128 Kbyte–256 Mbyte sizes
		Implemented with IBAT and DBAT registers in BAT array
Memory protection	Architecturally defined	Segments selectable as no-execute
		Pages selectable as user/supervisor and read-only or guarded
		Blocks selectable as user/supervisor and read-only or guarded
Page history	Architecturally defined	Referenced and changed bits defined and maintained

Table 5-1. MMU Feature Summary (Continued)

Feature Category	Architecturally Defined/ Gekko-Specific	Feature
Page address translation	Architecturally defined	Translations stored as PTEs in hashed page tables in memory
		Page table size determined by mask in SDR1 register
TLBs	Architecturally defined	Instructions for maintaining TLBs (**tlbie** and **tlbsync** instructions in Gekko)
	Gekko-specific	128-entry, two-way set associative ITLB 128-entry, two-way set associative DTLB LRU replacement algorithm
Segment descriptors	Architecturally defined	Stored as segment registers on-chip (two identical copies maintained)
Page table search support	Gekko-specific	Gekko performs the table search operation in hardware.

5.1.1 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a load, store, branch, or cache instruction, and when it fetches the next instruction. The effective address is translated to a physical address according to the procedures described in Chapter 7, “Memory Management” in the *PowerPC Microprocessor Family: The Programming Environments* manual, augmented with information in this chapter. The memory subsystem uses the physical address for the access.

For a complete discussion of effective address calculation, see Section 2.3.2.3 on Page 2-35.

5.1.2 MMU Organization

Figure 5-1 shows the conceptual organization of a PowerPC MMU in a 32-bit implementation; note that it does not describe the specific hardware used to implement the memory management function for a particular processor. Processors may optionally implement on-chip TLBs, hardware support for the automatic search of the page tables for PTEs, and other hardware features (invisible to the system software) not shown.

Gekko maintains two on-chip TLBs with the following characteristics:

- 128 entries, two-way set associative (64 x 2), LRU replacement
- Data TLB supports the DMMU; instruction TLB supports the IMMU
- Hardware TLB update
- Hardware update of referenced (R) and changed (C) bits in the translation table

In the event of a TLB miss, the hardware attempts to load the TLB based on the results of a translation table search operation.

Figure 5-2 and Figure 5-3 show the conceptual organization of Gekko’s instruction and data MMUs, respectively. The instruction addresses shown in Figure 5-2 are generated by the processor for sequential instruction fetches and addresses that correspond to a change of program flow. Data addresses shown in Figure 5-3 are generated by load, store, and cache instructions.

As shown in the figures, after an address is generated, the high-order bits of the effective address, EA[0–19] (or a smaller set of address bits, EA[0–*n*], in the cases of blocks), are translated into physical address bits PA[0–19]. The low-order address bits, A[20–31], are untranslated and are

therefore identical for both effective and physical addresses. After translating the address, the MMUs pass the resulting 32-bit physical address to the memory subsystem. The MMUs record whether the translation is for an instruction or data access, whether the processor is in user or supervisor mode and, for data accesses, whether the access is a load or a store operation.

The MMUs use this information to appropriately direct the address translation and to enforce the protection hierarchy programmed by the operating system. Section 4.3 on Page 4-7 describes the MSR, which controls some of the critical functionality of the MMUs.

The figures show how address bits A[20–26] index into the on-chip instruction and data caches to select a cache set. The remaining physical address bits are then compared with the tag fields (comprised of bits PA[0–19]) of the two selected cache blocks to determine if a cache hit has occurred. In the case of a cache miss on Gekko, the instruction or data access is then forwarded to the L2 tags to check for an L2 cache hit. In case of a miss the access is forwarded to the bus interface unit which initiates an external memory access.

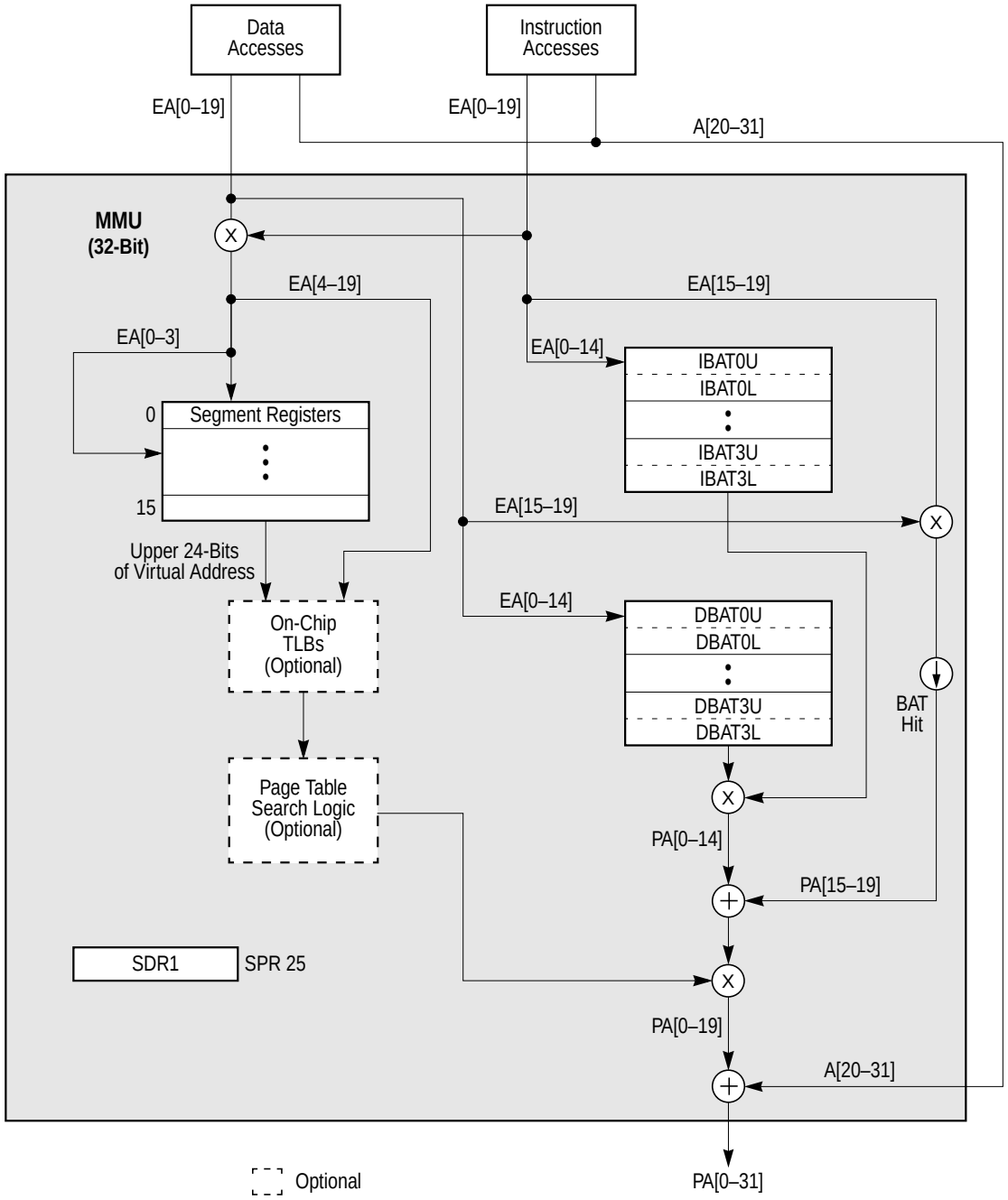


Figure 5-1. MMU Conceptual Block Diagram

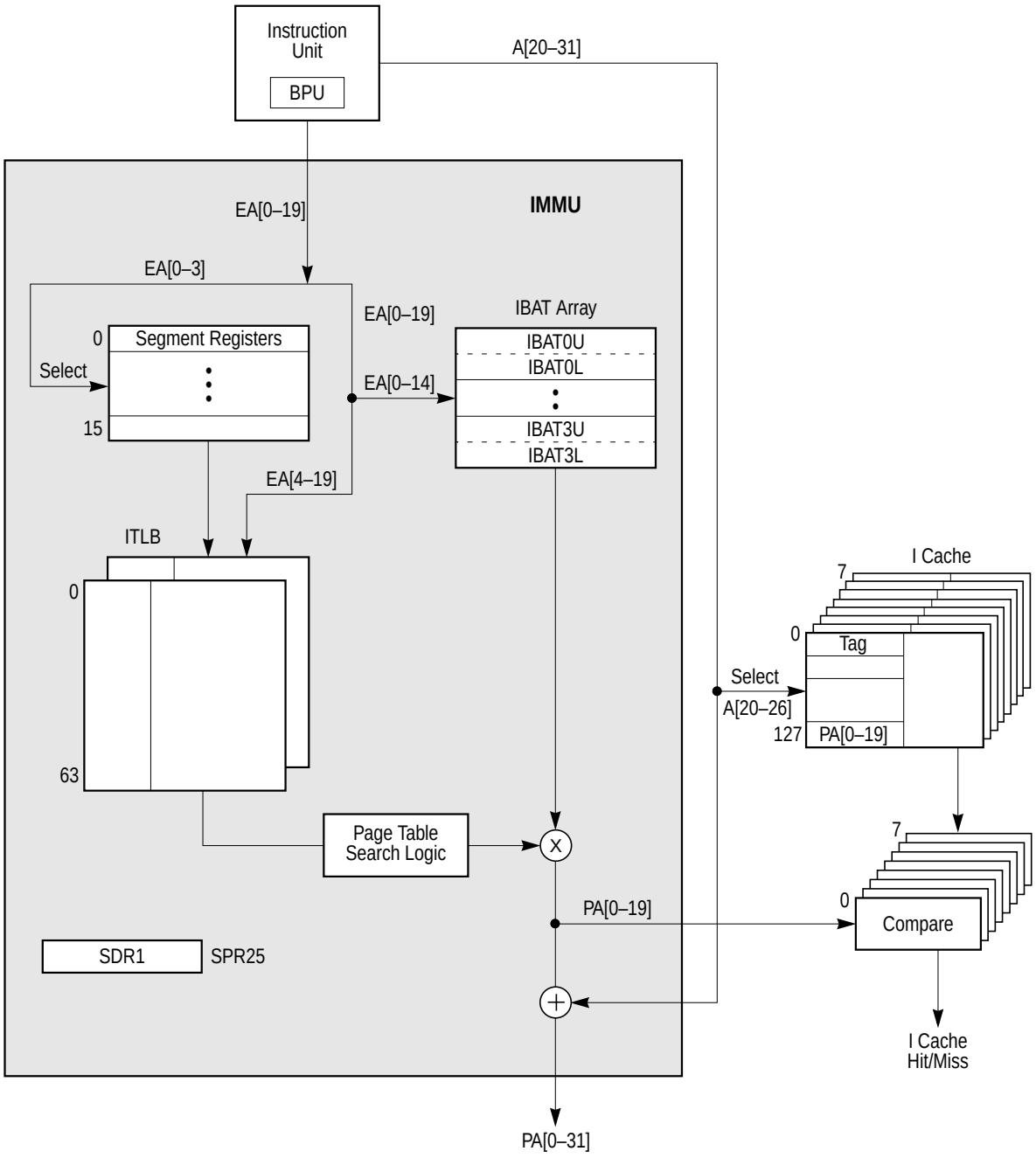


Figure 5-2. PowerPC Gekko Microprocessor IMMU Block Diagram

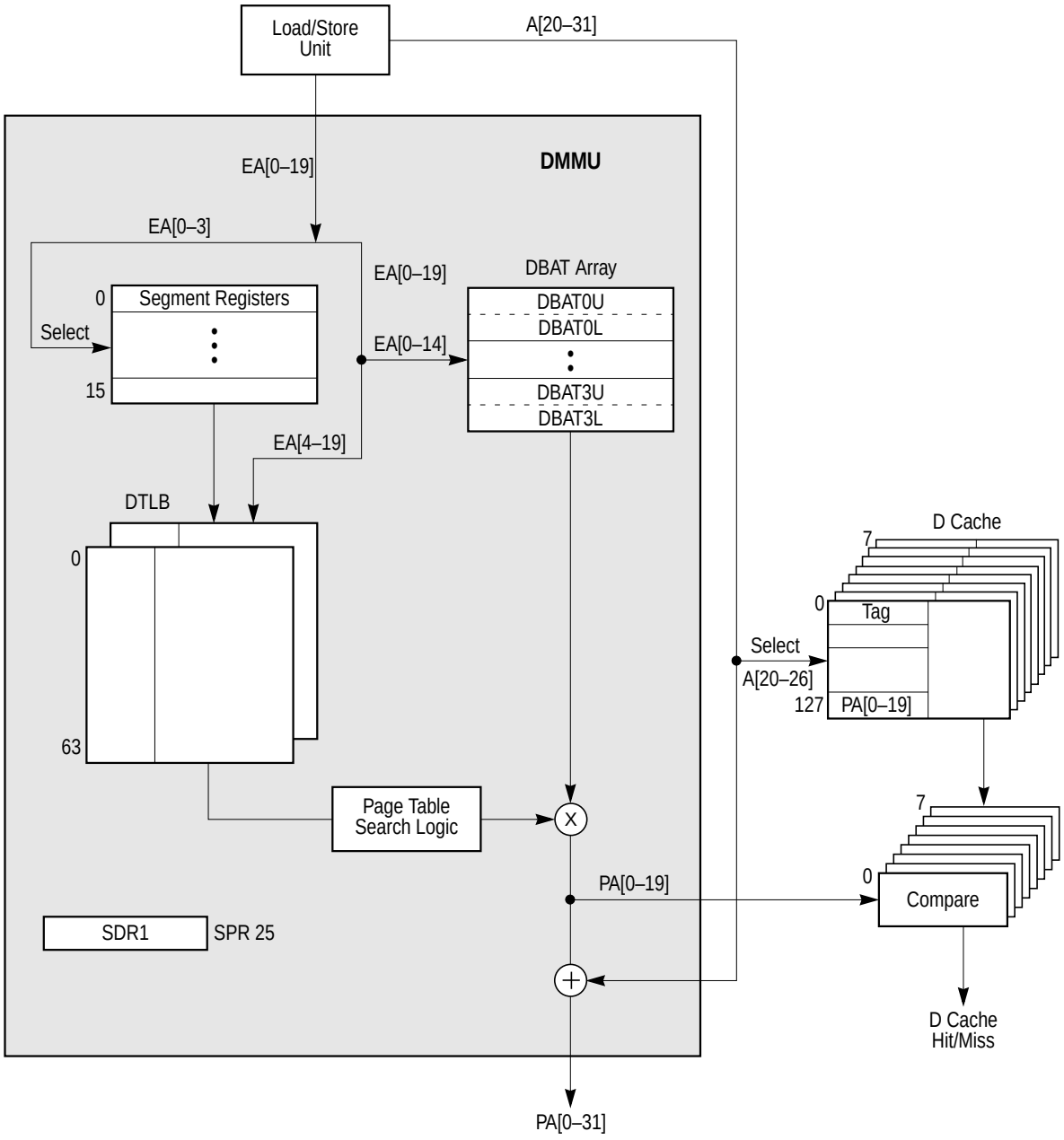


Figure 5-3. Gekko Microprocessor DMMU Block Diagram

5.1.3 Address Translation Mechanisms

PowerPC processors support the following three types of address translation:

- Page address translation—translates the page frame address for a 4-Kbyte page size
- Block address translation—translates the block number for blocks that range in size from 128 Kbytes to 256 Mbytes.
- Real addressing mode address translation—when address translation is disabled, the physical address is identical to the effective address.

Figure 5-4 shows the three address translation mechanisms provided by the MMUs. The segment descriptors shown in the figure control the page address translation mechanism. When an access uses page address translation, the appropriate segment descriptor is required. In 32-bit implementations, the appropriate segment descriptor is selected from the 16 on-chip segment registers by the four highest-order effective address bits.

A control bit in the corresponding segment descriptor then determines if the access is to memory (memory-mapped) or to the direct-store interface space. Note that the direct-store interface was present in the architecture only for compatibility with existing I/O devices that used this interface. However, it is being removed from the architecture, and Gekko does not support it. When an access is determined to be to the direct-store interface space, Gekko takes a DSI exception if it is a data access (see Section 4.5.3 on Page 4-17), and takes an ISI exception if it is an instruction access (see Section 4.5.4 on Page 4-18).

For memory accesses translated by a segment descriptor, the interim virtual address is generated using the information in the segment descriptor. Page address translation corresponds to the conversion of this virtual address into the 32-bit physical address used by the memory subsystem. In most cases, the physical address for the page resides in an on-chip TLB and is available for quick access. However, if the page address translation misses in the on-chip TLB, the MMU causes a search of the page tables in memory (using the virtual address information and a hashing function) to locate the required physical address.

Because blocks are larger than pages, there are fewer upper-order effective address bits to be translated into physical address bits (more low-order address bits (at least 17) are untranslated to form the offset into a block) for block address translation. Also, instead of segment descriptors and a TLB, block address translations use the on-chip BAT registers as a BAT array. If an effective address matches the corresponding field of a BAT register, the information in the BAT register is used to generate the physical address; in this case, the results of the page translation (occurring in parallel) are ignored.

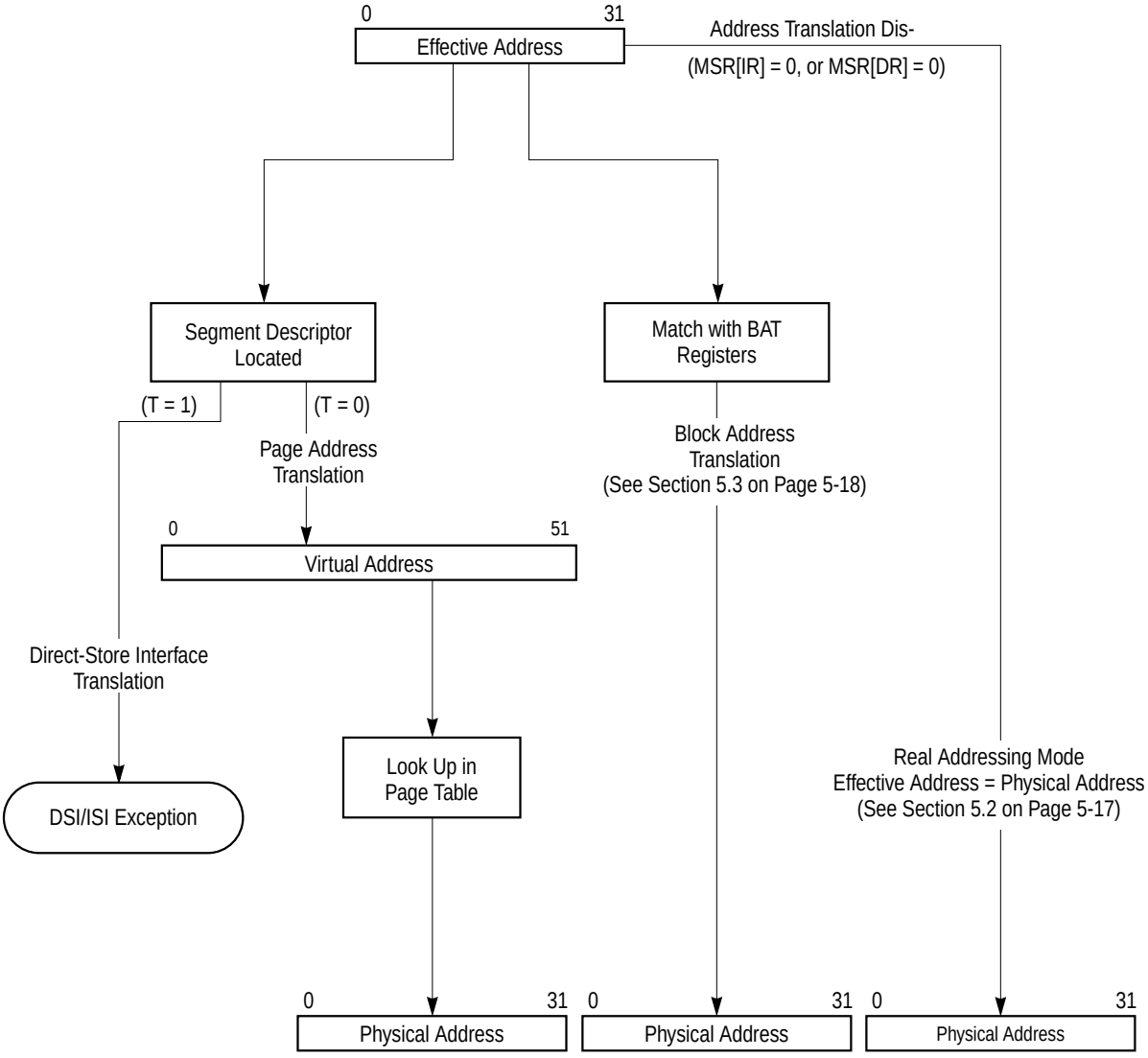


Figure 5-4. Address Translation Types

When the processor generates an access, and the corresponding address translation enable bit in MSR is cleared, the resulting physical address is identical to the effective address and all other translation mechanisms are ignored. Instruction address translation and data address translation are enabled by setting MSR[IR] and MSR[DR], respectively.

5.1.4 Memory Protection Facilities

In addition to the translation of effective addresses to physical addresses, the MMUs provide access protection of supervisor areas from user access and can designate areas of memory as read-only as well as no-execute or guarded. Table 5-2 shows the protection options supported by the MMUs for pages.

Table 5-2. Access Protection Options for Pages

Option	User Read		User Write	Supervisor Read		Supervisor Write
	I-Fetch	Data		I-Fetch	Data	
Supervisor-only	—	—	—			
Supervisor-only-no-execute	—	—	—	—		
Supervisor-write-only			—			
Supervisor-write-only-no-execute	—		—	—		
Both (user/supervisor)						
Both (user-/supervisor) no-execute	—			—		
Both (user-/supervisor) read-only			—			—
Both (user/supervisor) read-only-no-execute	—		—	—		—
Access permitted — Protection violation						

The no-execute option provided in the segment register lets the operating system program determine whether instructions can be fetched from an area of memory. The remaining options are enforced based on a combination of information in the segment descriptor and the page table entry. Thus, the supervisor-only option allows only read and write operations generated while the processor is operating in supervisor mode (MSR[PR] = 0) to access the page. User accesses that map into a supervisor-only page cause an exception.

Finally, a facility in the VEA and OEA allows pages or blocks to be designated as guarded, preventing out-of-order accesses that may cause undesired side effects. For example, areas of the memory map used to control I/O devices can be marked as guarded so accesses do not occur unless they are explicitly required by the program.

For more information on memory protection, see “Memory Protection Facilities,” in Chapter 7, “Memory Management,” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

5.1.5 Page History Information

The MMUs of PowerPC processors also define referenced (R) and changed (C) bits in the page address translation mechanism that can be used as history information relevant to the page. The operating system can use these bits to determine which areas of memory to write back to disk when new pages must be allocated in main memory. While these bits are initially programmed by the operating system into the page table, the architecture specifies that they can be maintained either by the processor hardware (automatically) or by some software-assist mechanism.

Implementation Note—When loading the TLB, Gekko checks the state of the changed and referenced bits for the matched PTE. If the referenced bit is not set and the table search operation is initially caused by a load operation or by an instruction fetch, Gekko automatically sets the referenced bit in the translation table. Similarly, if the table search operation is caused by a store operation and either the referenced bit or the changed bit is not set, the hardware automatically sets both bits in the translation table. In addition, when the address translation of a store operation hits in the DTLB, Gekko checks the state of the changed bit. If the bit is not already set, the hardware automatically updates the DTLB and the translation table in memory to set the changed bit. For more information, see Section 5.4.1 on Page 5-18.

5.1.6 General Flow of MMU Address Translation

The following sections describe the general flow used by PowerPC processors to translate effective addresses to virtual and then physical addresses.

5.1.6.1 Real Addressing Mode and Block Address Translation Selection

When an instruction or data access is generated and the corresponding instruction or data translation is disabled (MSR[IR] = 0 or MSR[DR] = 0), real addressing mode is used (physical address equals effective address) and the access continues to the memory subsystem as described in Section 5.2 on Page 5-17.

Figure 5-5 shows the flow the MMUs use in determining whether to select real addressing mode, block address translation, or the segment descriptor to select page address translation.

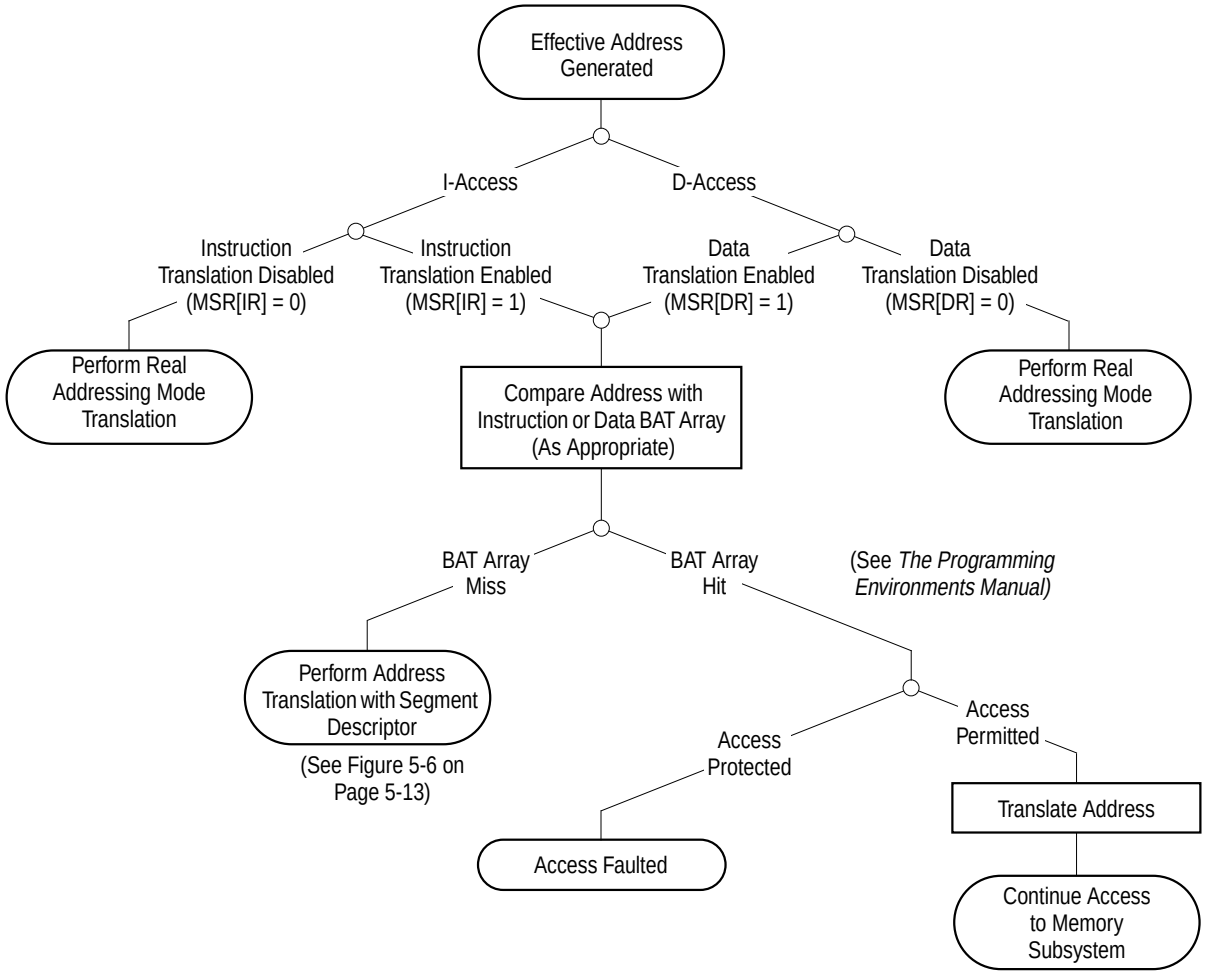


Figure 5-5. General Flow of Address Translation (Real Addressing Mode and Block)

NOTE: If the BAT array search results in a hit, the access is qualified with the appropriate protection bits. If the access violates the protection mechanism, an exception (either ISI or DSI) is generated.

5.1.6.2 Page Address Translation Selection

If address translation is enabled and the effective address information does not match a BAT array entry, the segment descriptor must be located. When the segment descriptor is located, the T bit in the segment descriptor selects whether the translation is to a page or to a direct-store segment as shown in Figure 5-6 on Page 5-13.

For 32-bit implementations, the segment descriptor for an access is contained in one of 16 on-chip segment registers; effective address bits EA[0–3] select one of the 16 segment registers.

Note that Gekko does not implement the direct-store interface, and accesses to these segments cause a DSI or ISI exception. In addition, Figure 5-6 also shows the way in which the no-execute protection is enforced; if the N bit in the segment descriptor is set and the access is an instruction fetch, the access is faulted as described in Chapter 7, “Memory Management,” in the *PowerPC Microprocessor Family: The Programming Environments* manual. Note that the figure shows the flow for these cases as described by the PowerPC OEA, and so the TLB references are shown as optional. Because Gekko

implements TLBs, these branches are valid and are described in more detail throughout this chapter.

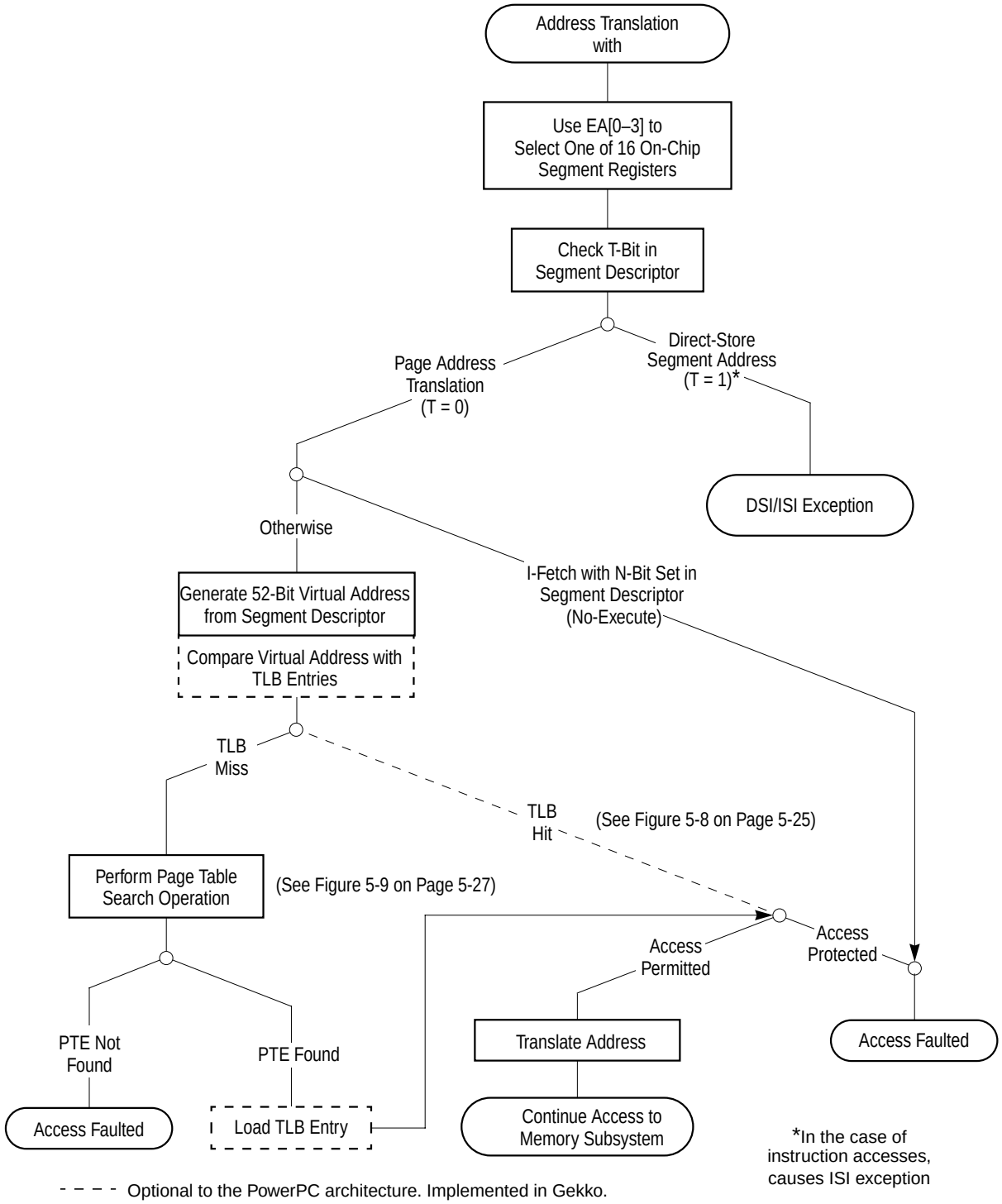


Figure 5-6. General Flow of Page and Direct-Store Interface Address Translation

If SR[T] = 0, page address translation is selected. The information in the segment descriptor is then used to generate the 52-bit virtual address. The virtual address is then used to identify the page address translation information (stored as page table entries (PTEs) in a page table in memory). For increased performance, Gekko has two on-chip TLBs to cache recently-used translations on-chip.

If an access hits in the appropriate TLB, page translation succeeds and the physical address bits are forwarded to the memory subsystem. If the required translation is not resident, the MMU performs a search of the page table. If the required PTE is found, a TLB entry is allocated and the page translation is attempted again. This time, the TLB is guaranteed to hit. When the translation is located, the access is qualified with the appropriate protection bits. If the access causes a protection violation, either an ISI or DSI exception is generated.

If the PTE is not found by the table search operation, a page fault condition exists, and an ISI or DSI exception occurs so software can handle the page fault.

5.1.7 MMU Exceptions Summary

To complete any memory access, the effective address must be translated to a physical address. As specified by the architecture, an MMU exception condition occurs if this translation fails for one of the following reasons:

- Page fault—there is no valid entry in the page table for the page specified by the effective address (and segment descriptor) and there is no valid BAT translation.
- An address translation is found but the access is not allowed by the memory protection mechanism.

The translation exception conditions defined by the OEA for 32-bit implementations cause either the ISI or the DSI exception to be taken as shown in Table 5-3.I

Table 5-3. Translation Exception Conditions

Condition	Description	Exception
Page fault (no PTE found)	No matching PTE found in page tables (and no matching BAT array entry)	I access: ISI exception SRR1[1] = 1
		D access: DSI exception DSISR[1] =1
Block protection violation	Conditions described for block in “Block Memory Protection” in Chapter 7, “Memory Management,” in the *PowerPC Microprocessor Family: The Programming Environments* manual.“	I access: ISI exception SRR1[4] = 1
		D access: DSI exception DSISR[4] =1
Page protection violation	Conditions described for page in “Page Memory Protection” in Chapter 7, “Memory Management,” in the *PowerPC Microprocessor Family: The Programming Environments* manual.	I access: ISI exception SRR1[4] = 1
		D access: DSI exception DSISR[4] =1
No-execute protection violation	Attempt to fetch instruction when SR[N] = 1	ISI exception SRR1[3] = 1
Instruction fetch from direct-store segment	Attempt to fetch instruction when SR[T] = 1	ISI exception SRR1[3] =1

Table 5-3. Translation Exception Conditions (Continued)

Condition	Description	Exception
Data access to direct-store segment (including floating-point accesses)	Attempt to perform load or store (including FP load or store) when SR[T] = 1	DSI exception DSISR[5] =1
Instruction fetch from guarded memory	Attempt to fetch instruction when MSR[IR] = 1 and either matching xBAT[G] = 1, or no matching BAT entry and PTE[G] = 1	ISI exception SRR1[3] =1

The state saved by the processor for each of these exceptions contains information that identifies the address of the failing instruction. Refer to Chapter 4, "Exceptions" in this manual for a more detailed description of exception processing.

In addition to the translation exceptions, there are other MMU-related conditions (some of them defined as implementation-specific, and therefore not required by the architecture) that can cause an exception to occur.

These exception conditions map to processor exceptions as shown in Table 5-4 on Page 5-15. The only MMU exception conditions that occur when MSR[DR] = 0 are those that cause an alignment exception for data accesses. For more detailed information about the conditions that cause an alignment exception (in particular for string/multiple instructions), see Section 4.5.6 on Page 4-19.

NOTE: Some exception conditions depend upon whether the memory area is set up as write-though (W = 1) or cache-inhibited (I = 1).
These bits are described fully in “Memory/Cache Access Attributes,” in Chapter 5, “Cache Model and Memory Coherency,” of the *PowerPC Microprocessor Family: The Programming Environments* manual.
Also refer to Chapter 4, "Exceptions" in this manual and to Chapter 6, “Exceptions,” in the *PowerPC Microprocessor Family: The Programming Environments* manual for a complete description of the SRR1 and DSISR bit settings for these exceptions.

Table 5-4. Other MMU Exception Conditions for the Gekko Processor

Condition	Description	Exception
dcbz or **dcbz_l** with W = 1 or I = 1	**dcbz** or **dcbz_l** instruction to write-through or cache-inhibited segment or block	Alignment exception (not required by architecture for this condition)
lwarx or **stwcx.** with W = 1	Reservation instruction to write-through segment or block	DSI exception DSISR[5] =1
lwarx, **stwcx.**, **eciwx**, or **ecowx** instruction to direct-store segment	Reservation instruction or external control instruction when SR[T] =1	DSI exception DSISR[5] =1
Floating-point load or store to direct-store segment	FP memory access when SR[T] =1	See data access to direct-store segment in Table 5-4 on Page 5-15.
Load or store that results in a direct-store error	Does not occur in 750	Does not apply
eciwx or **ecowx** attempted when external control facility disabled	**eciwx** or **ecowx** attempted with EAR[E] = 0	DSI exception DSISR[11] = 1

Table 5-4. Other MMU Exception Conditions for the Gekko Processor

Condition	Description	Exception
lmw, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx** instruction attempted in little-endian mode	**lmw**, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx** instruction attempted while MSR[LE] = 1	Alignment exception
Operand misalignment	Translation enabled and a floating-point load/store, **stmw**, **stwcx.**, **lmw**, **lwarx**, **eciwx**, or **ecowx** instruction operand is not word-aligned	Alignment exception (some of these cases are implementation-specific)

5.1.8 MMU Instructions and Register Summary

The MMU instructions and registers allow the operating system to set up the block address translation areas and the page tables in memory.

NOTE: Because the implementation of TLBs is optional, the instructions that refer to these structures are also optional. However, as these structures serve as caches of the page table, the architecture specifies a software protocol for maintaining coherency between these caches and the tables in memory whenever the tables in memory are modified. When the tables in memory are changed, the operating system purges these caches of the corresponding entries, allowing the translation caching mechanism to refetch from the tables when the corresponding entries are required.
Also note that Gekko implements all TLB-related instructions except **tlbia**, which is treated as an illegal instruction.

Because the MMU specification for PowerPC processors is so flexible, it is recommended that the software that uses these instructions and registers be encapsulated into subroutines to minimize the impact of migrating across the family of implementations.

Table 5-5 summarizes Gekko's instructions that specifically control the MMU. For more detailed information about the instructions, refer to Chapter 2, "Programming Model" in this manual and Chapter 8, "Instruction Set," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Table 5-5. Gekko Microprocessor Instruction Summary—Control MMUs

Instruction	Description
mtsr SR,**r**S	Move to Segment Register SR[SR#]← **r**S
mtsrin rS,**r**B	Move to Segment Register Indirect SR[**r**B[0–3]]←**r**S
mfsr rD,SR	Move from Segment Register **r**D←SR[SR#]
mfsrin rD,**r**B	Move from Segment Register Indirect **r**D←SR[**r**B[0–3]]

Table 5-5. Gekko Microprocessor Instruction Summary—Control MMUs

Instruction	Description
tlbie rB*	TLB Invalidate Entry For effective address specified by **r**B, TLB[V]←0 The **tlbie** instruction invalidates all TLB entries indexed by the EA, and operates on both the instruction and data TLBs simultaneously invalidating four TLB entries. The index corresponds to bits 14–19 of the EA. Software must ensure that instruction fetches or memory references to the virtual pages specified by the **tlbie** instruction have been completed prior to executing the **tlbie** instruction.
tlbsync*	TLB Synchronize Synchronizes the execution of all other **tlbie** instructions in the system. In Gekko, when the $\overline{\text{TLBISYNC}}$ signal is negated, instruction execution may continue or resume after the completion of a **tlbsync** instruction. When the $\overline{\text{TLBISYNC}}$ signal is asserted, instruction execution stops after the completion of a **tlbsync** instruction.
*These instructions are defined by the PowerPC architecture, but are optional.	

Table 5-6 summarizes the registers that the operating system uses to program Gekko's MMUs. These registers are accessible to supervisor-level software only.

These registers are described in Chapter 2, "Programming Model" in this manual.

Table 5-6. Gekko Microprocessor MMU Registers

Register	Description
Segment registers (SR0–SR15)	The sixteen 32-bit segment registers are present only in 32-bit implementations of the PowerPC architecture. The fields in the segment register are interpreted differently depending on the value of bit 0. The segment registers are accessed by the **mtsr**, **mtsrin**, **mfsr**, and **mfsrin** instructions.
BAT registers (IBAT0U–IBAT3U, IBAT0L–IBAT3L, DBAT0U–DBAT3U, and DBAT0L–DBAT3L)	There are 16 BAT registers, organized as four pairs of instruction BAT registers (IBAT0U–IBAT3U paired with IBAT0L–IBAT3L) and four pairs of data BAT registers (DBAT0U–DBAT3U paired with DBAT0L–DBAT3L). The BAT registers are defined as 32-bit registers in 32-bit implementations. These are special-purpose registers that are accessed by the **mtspr** and **mfspr** instructions.
SDR1	The SDR1 register specifies the variables used in accessing the page tables in memory. SDR1 is defined as a 32-bit register for 32-bit implementations. This special-purpose register is accessed by the **mtspr** and **mfspr** instructions.

5.2 Real Addressing Mode

If address translation is disabled (MSR[IR] = 0 or MSR[DR] = 0) for a particular access, the effective address is treated as the physical address and is passed directly to the memory subsystem as described in Chapter 7, "Memory Management," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Note that the default WIMG bits (0b0011) cause data accesses to be considered cacheable (I = 0) and thus load and store accesses are weakly ordered. This is the case even if the data cache is disabled in the HID0 register (as it is out of hard reset). If I/O devices require load and store accesses to occur in strict program order (strongly ordered), translation must be enabled so that the corresponding I bit can be set. Note also, that the G bit must be set to ensure that the accesses are strongly ordered. For instruction accesses, the default memory access mode bits (WIMG) are also

0b0011. That is, instruction accesses are considered cacheable (I = 0), and the memory is guarded. Again, instruction accesses are considered cacheable even if the instruction cache is disabled in the HID0 register (as it is out of hard reset). The W and M bits have no effect on the instruction cache.

For information on the synchronization requirements for changes to MSR[IR] and MSR[DR], refer to Section 2.3.2.4 on Page 2-36 in this manual and the section "Synchronization Requirements for Special Registers and for Lookaside Buffers" in Chapter 2 of the *PowerPC Microprocessor Family: The Programming Environments* manual.

5.3 Block Address Translation

The block address translation (BAT) mechanism in the OEA provides a way to map ranges of effective addresses larger than a single page into contiguous areas of physical memory. Such areas can be used for data that is not subject to normal virtual memory handling (paging), such as a memory-mapped display buffer or an extremely large array of numerical data.

Block address translation in Gekko is described in Chapter 7, "Memory Management," in the *PowerPC Microprocessor Family: The Programming Environments* manual for 32-bit implementations.

Implementation Note—Gekko's BAT registers are not initialized by the hardware after the power-up or reset sequence. Consequently, all valid bits in both instruction and data BATs must be cleared before setting any BAT for the first time. This is true regardless of whether address translation is enabled. Also, software must avoid overlapping blocks while updating a BAT or areas. **Even if translation is disabled, multiple BAT hits are treated as programming errors and can corrupt the BAT registers and produce unpredictable results. Always re-zero during the reset ISR. After zeroing all BATs, set them (in order) to the desired values. $\overline{\text{HRESET}}$ disorders the BATs. $\overline{\text{SRESET}}$ does not.**

5.4 Memory Segment Model

Gekko adheres to the memory segment model as defined in Chapter 7, "Memory Management," in the *PowerPC Microprocessor Family: The Programming Environments* manual for 32-bit implementations. Memory in the PowerPC OEA is divided into 256-Mbyte segments. This segmented memory model provides a way to map 4-Kbyte pages of effective addresses to 4-Kbyte pages in physical memory (page address translation), while providing the programming flexibility afforded by a large virtual address space (52 bits).

The segment/page address translation mechanism may be superseded by the block address translation (BAT) mechanism described Section 5.3. If not, the translation proceeds in the following two steps:

1. from effective address to the virtual address (which never exists as a specific entity but can be considered to be the concatenation of the virtual page number and the byte offset within a page), and
2. from virtual address to physical address.

This section highlights those areas of the memory segment model defined by the OEA that are specific to Gekko.

5.4.1 Page History Recording

Referenced (R) and changed (C) bits in each PTE keep history information about the page. They are maintained by a combination of Gekko's table search hardware and the system software. The operating system uses this information to determine which areas of memory to write back to disk when new pages must be allocated in main memory. Referenced and changed recording is performed

only for accesses made with page address translation and not for translations made with the BAT mechanism or for accesses that correspond to direct-store (T = 1) segments. Furthermore, R and C bits are maintained only for accesses made while address translation is enabled (MSR[IR] = 1 or MSR[DR] = 1).

In Gekko, the referenced and changed bits are updated as follows:

- For TLB hits, the C bit is updated according to Table 5-7.
- For TLB misses, when a table search operation is in progress to locate a PTE. The R and C bits are updated (set, if required) to reflect the status of the page based on this access.

Table 5-7. Table Search Operations to Update History Bits—TLB Hit Case

R and C bits in TLB Entry	Processor Action
00	Combination doesn't occur
01	Combination doesn't occur
10	Read: No special action Write: Gekko initiates a table search operation to update C.
11	No special action for read or write

The table shows that the status of the C bit in the TLB entry (in the case of a TLB hit) is what causes the processor to update the C bit in the PTE (the R bit is assumed to be set in the page tables if there is a TLB hit). Therefore, when software clears the R and C bits in the page tables in memory, it must invalidate the TLB entries associated with the pages whose referenced and changed bits were cleared.

The **dcbt** and **dcbtst** instructions can execute if there is a TLB/BAT hit or if the processor is in real addressing mode. In case of a TLB or BAT miss, these instructions are treated as no-ops; they do not initiate a table search operation and they do not set either the R or C bits.

As defined by the PowerPC architecture, the referenced and changed bits are updated as if address translation were disabled (real addressing mode). If these update accesses hit in the data cache, they are not seen on the external bus. If they miss in the data cache, they are performed as typical cache line fill accesses on bus (assuming the data cache is enabled).

5.4.1.1 Referenced Bit

The referenced (R) bit of a page is located in the PTE in the page table. Every time a page is referenced (with a read or write access) and the R bit is zero, Gekko sets the R bit in the page table. The OEA specifies that the referenced bit may be set immediately, or the setting may be delayed until the memory access is determined to be successful. Because the reference to a page is what causes a PTE to be loaded into the TLB, the referenced bit in all TLB entries is effectively always set. The processor never automatically clears the referenced bit.

The referenced bit is only a hint to the operating system about the activity of a page. At times, the referenced bit may be set although the access was not logically required by the program or even if the access was prevented by memory protection. Examples of this in PowerPC systems include the following:

- Fetching of instructions not subsequently executed
- A memory reference caused by a speculatively executed instruction that is mispredicted

- Accesses generated by an **lswx** or **stswx** instruction with a zero length
- Accesses generated by an **stwcx.** instruction when no store is performed because a reservation does not exist
- Accesses that cause exceptions and are not completed

5.4.1.2 Changed Bit

The changed bit of a page is located both in the PTE in the page table and in the copy of the PTE loaded into the TLB (if a TLB is implemented, as in Gekko). Whenever a data store instruction is executed successfully, if the TLB search (for page address translation) results in a hit, the changed bit in the matching TLB entry is checked. If it is already set, it is not updated. If the TLB changed bit is 0, Gekko initiates the table search operation to set the C bit in the corresponding PTE in the page table. Gekko then reloads the TLB (with the C bit set).

The changed bit (in both the TLB and the PTE in the page tables) is set only when a store operation is allowed by the page memory protection mechanism and the store is guaranteed to be in the execution path (unless an exception, other than those caused by the **sc**, **rfi**, or trap instructions, occurs). Furthermore, the following conditions may cause the C bit to be set:

- The execution of an **stwcx.** instruction is allowed by the memory protection mechanism but a store operation is not performed.
- The execution of an **stswx** instruction is allowed by the memory protection mechanism but a store operation is not performed because the specified length is zero.
- The store operation is not performed because an exception occurs before the store is performed.

Again, note that although the execution of the **dcbt** and **dcbtst** instructions may cause the R bit to be set, they never cause the C bit to be set.

5.4.1.3 Scenarios for Referenced and Changed Bit Recording

This section provides a summary of the model (defined by the OEA) that is used by PowerPC processors for maintaining the referenced and changed bits. In some scenarios, the bits are guaranteed to be set by the processor, in some scenarios, the architecture allows that the bits may be set (not absolutely required), and in some scenarios, the bits are guaranteed to not be set. Note that when Gekko updates the R and C bits in memory, the accesses are performed as if MSR[DR] = 0 and G = 0 (that is, as nonguarded cacheable operations in which coherency is required).

Table 5-8 defines a prioritized list of the R and C bit settings for all scenarios. The entries in the table are prioritized from top to bottom, such that a matching scenario occurring closer to the top of the table takes precedence over a matching scenario closer to the bottom of the table. For example, if an **stwcx.** instruction causes a protection violation and there is no reservation, the C bit is not altered, as shown for the protection violation case. Note that in the table, load operations include those generated by load instructions, by the **eciwx** instruction, and by the cache management instructions that are treated as a load with respect to address translation. Similarly, store operations include those operations generated by store instructions, by the **ecowx** instruction, and by the cache management

instructions that are treated as a store with respect to address translation.

Table 5-8. Model for Guaranteed R and C Bit Settings

Priority	Scenario	Causes Setting of R Bit		Causes Setting of C Bit	
		OEA	Gekko	OEA	Gekko
1	No-execute protection violation	No	No	No	No
2	Page protection violation	Maybe	Yes	No	No
3	Out-of-order instruction fetch or load operation	Maybe	No	No	No
4	Out-of-order store operation. Would be required by the sequential execution model in the absence of system-caused or imprecise exceptions, or of floating-point assist exception for instructions that would cause no other kind of precise exception.	Maybe[1]	No	No	No
5	All other out-of-order store operations	Maybe[1]	No	Maybe[1]	No
6	Zero-length load (**lswx**)	Maybe	No	No	No
7	Zero-length store (**stswx**)	Maybe[1]	No	Maybe[1]	No
8	Store conditional (**stwcx.**) that does not store	Maybe[1]	Yes	Maybe[1]	Yes
9	In-order instruction fetch	Yes	Yes	No	No
10	Load instruction or **eciwx**	Yes	Yes	No	No
11	Store instruction, **ecowx, dcbz_l** or **dcbz** instruction	Yes	Yes	Yes	Yes
12	**icbi**, **dcbt**, or **dcbtst** instruction	Maybe	No	No	No
13	**dcbst** or **dcbf** instruction	Maybe	Yes	No	No
14	**dcbi** instruction	Maybe[1]	Yes	Maybe[1]	Yes

Notes:
[1] If C is set, R is guaranteed to be set also.

For more information, see “Page History Recording” in Chapter 7, “Memory Management,” of the *PowerPC Microprocessor Family: The Programming Environments* manual.

5.4.2 Page Memory Protection

Gekko implements page memory protection as it is defined in Chapter 7, “Memory Management,” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

5.4.3 TLB Description

Gekko implements separate 128-entry data and instruction TLBs to maximize performance. This section describes the hardware resources provided in Gekko to facilitate page address translation. Note that the hardware implementation of the MMU is not specified by the architecture, and while this description applies to Gekko, it does not necessarily apply to other PowerPC processors.

5.4.3.1 TLB Organization

Because Gekko has two MMUs (IMMU and DMMU) that operate in parallel, some of the MMU resources are shared, and some are actually duplicated (shadowed) in each MMU to maximize performance. For example, although the architecture defines a single set of segment registers for the MMU, Gekko maintains two identical sets of segment registers, one for the IMMU and one for the DMMU; when an instruction that updates the segment register executes, Gekko automatically updates both sets.

Each TLB contains 128 entries organized as a two-way set-associative array with 64 sets as shown in Figure 5-7 for the DTLB (the ITLB organization is the same). When an address is being translated, a set of two TLB entries is indexed in parallel with the access to a segment register. If the address in one of the two TLB entries is valid and matches the 40-bit virtual page number, that TLB entry contains the translation. If no match is found, a TLB miss occurs.

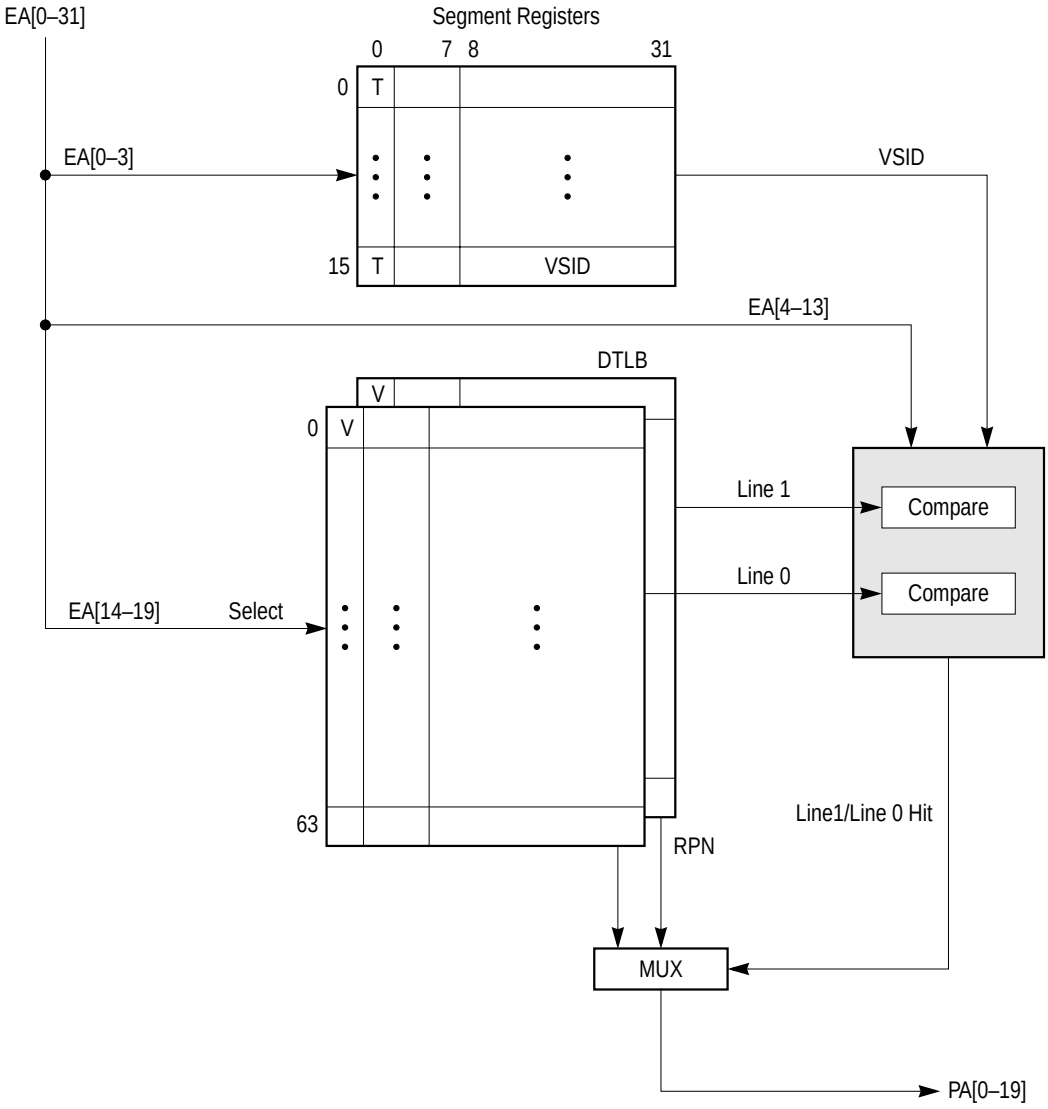


Figure 5-7. Segment Register and DTLB Organization

Unless the access is the result of an out-of-order access, a hardware table search operation begins if there is a TLB miss. If the access is out of order, the table search operation is postponed until the access is required, at which point the access is no longer out of order. When the matching PTE is found in memory, it is loaded into the TLB entry selected by the least-recently-used (LRU) replacement algorithm, and the translation process begins again, this time with a TLB hit.

To uniquely identify a TLB entry as the required PTE, the TLB entry also contains four more bits of the page index, EA[10–13] (in addition to the API bits in of the PTE).

Software cannot access the TLB arrays directly, except to invalidate an entry with the **tlbie** instruction. Each set of TLB entries has one associated LRU bit. The LRU bit for a set is updated any time either entry is used, even if the access is speculative. Invalid entries are always the first to be replaced.

Although both MMUs can be accessed simultaneously (both sets of segment registers and TLBs can be accessed in the same clock), only one exception condition can be reported at a time. ITLB miss exception conditions are reported when there are no more instructions to be dispatched or retired (the pipeline is empty), and DTLB miss conditions are reported when the load or store instruction is ready to be retired. Refer to Chapter 6, "Instruction Timing" in this manual for more detailed information about the internal pipelines and the reporting of exceptions.

When an instruction or data access occurs, the effective address is routed to the appropriate MMU. EA0–EA3 select one of the 16 segment registers and the remaining effective address bits and the VSID field from the segment register is passed to the TLB. EA[14–19] then select two entries in the TLB; the valid bits are checked and the 40-bit virtual page number (24-bit VSID and EA4–EA19]) must match the VSID, EAPI, and API fields of the TLB entries. If one of the entries hits, the PP bits are checked for a protection violation. If these bits don’t cause an exception, the C bit is checked and a table search operation is initiated if C must be updated. If C does not require updating, the RPN value is passed to the memory subsystem and the WIMG bits are then used as attributes for the access.

Although address translation is disabled on a reset condition, the valid bits of TLB entries are not automatically cleared. Thus, TLB entries must be explicitly cleared by the system software (with the **tlbie** instruction) before the valid entries are loaded and address translation is enabled. Also, note that the segment registers do not have a valid bit, and so they should also be initialized before translation is enabled.

5.4.3.2 TLB Invalidation

Gekko implements the optional **tlbie** and **tlbsync** instructions, which are used to invalidate TLB entries. The execution of the **tlbie** instruction always invalidates four entries—both the ITLB and DTLB entries indexed by EA[14–19].

The architecture allows **tlbie** to optionally enable a TLB invalidate signaling mechanism in hardware so that other processors also invalidate their resident copies of the matching PTE. Gekko does not signal the TLB invalidation to other processors nor does it perform any action when a TLB invalidation is performed by another processor.

The **tlbsync** instruction causes instruction execution to stop if the $\overline{\text{TLBISYNC}}$ signal is asserted. If $\overline{\text{TLBISYNC}}$ is negated, instruction execution may continue or resume after the completion of a **tlbsync** instruction. Section 8.9.2 on Page 8-38 describes the TLB synchronization mechanism in further detail.

The **tlbia** instruction is not implemented on Gekko and when its opcode is encountered, an illegal instruction program exception is generated. To invalidate all entries of both TLBs, 64 **tlbie** instructions must be executed, incrementing the value in EA14–EA19 by one each time.

(See Chapter 8, "Instruction Set" in the the *PowerPC Microprocessor Family: The Programming Environments* manual for detailed information about this instruction.)

Software must ensure that instruction fetches or memory references to the virtual pages specified by the **tlbie** have been completed prior to executing the **tlbie** instruction.

Other than the possible TLB miss on the next instruction prefetch, the **tlbie** instruction does not affect the instruction fetch operation—that is, the prefetch buffer is not purged and does not cause these instructions to be refetched.

5.4.4 Page Address Translation Summary

Figure 5-8 on Page 5-25 provides the detailed flow for the page address translation mechanism.

The figure includes the checking of the N bit in the segment descriptor and then expands on the ‘TLB Hit’ branch of Figure 5-6 on Page 5-13.

The detailed flow for the ‘TLB Miss’ branch of Figure 5-6 is described in Section 5.4.5 on Page 5-26.

NOTE: As in the case of block address translation, if an attempt is made to execute a **dcbz** or **dcbz_l** instruction to a page marked either write-through or caching-inhibited (W = 1 or I = 1), an alignment exception is generated. The checking of memory protection violation conditions is described in Chapter 7, “Memory Management” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

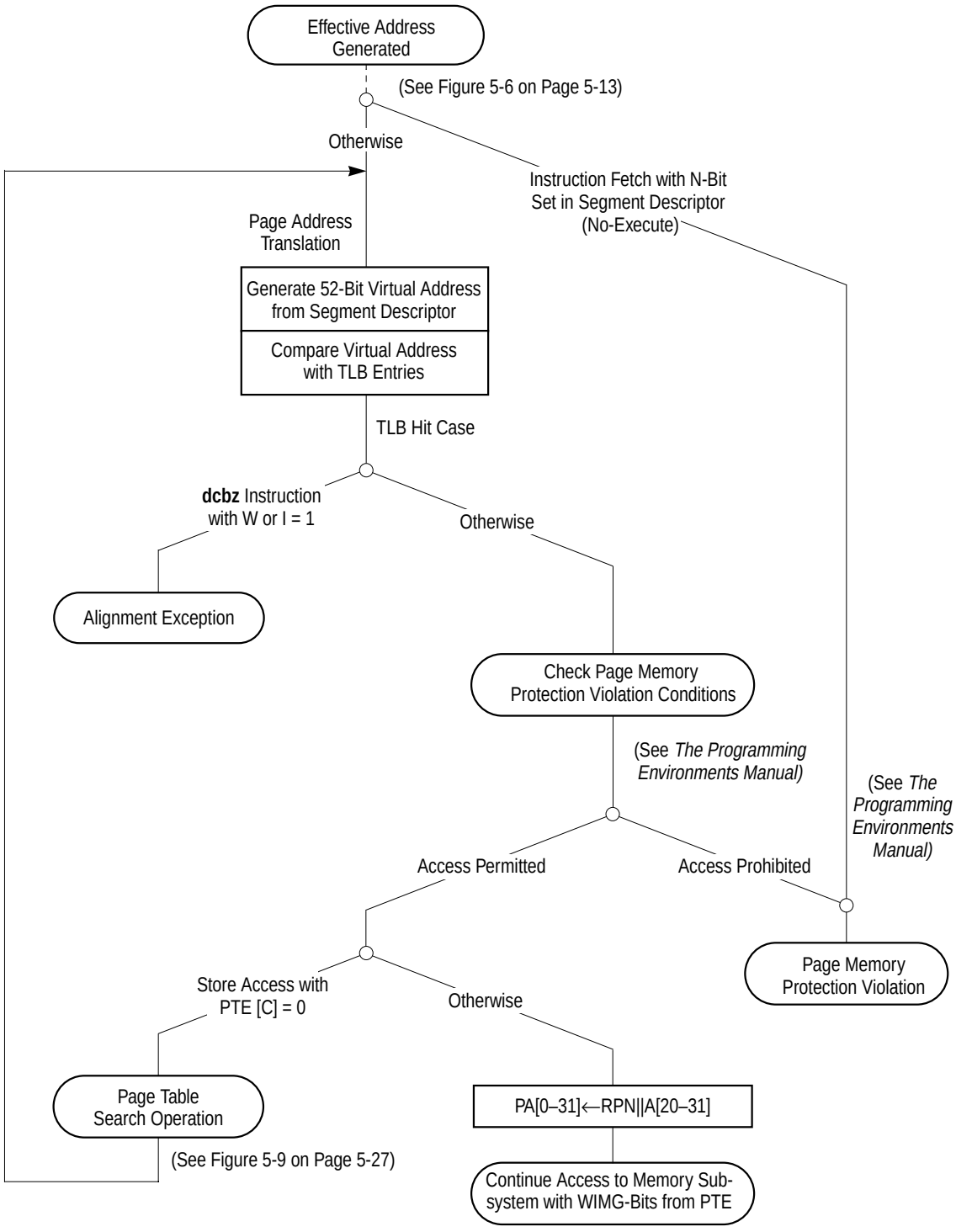


Figure 5-8. Page Address Translation Flow—TLB Hit

5.4.5 Page Table Search Operation

If the translation is not found in the TLBs (a TLB miss), Gekko initiates a table search operation which is described in this section. Formats for the PTE are given in "PTE Format for 32-Bit Implementations," in Chapter 7, "Memory Management" of the *PowerPC Microprocessor Family: The Programming Environments* manual.

The following is a summary of the page table search process performed by Gekko:

1. The 32-bit physical address of the primary PTEG is generated as described in "Page Table Addresses" in Chapter 7, "Memory Management" of the *PowerPC Microprocessor Family: The Programming Environments* manual.
2. The first PTE (PTE0) in the primary PTEG is read from memory. PTE reads occur with an implied WIM memory/cache mode control bit setting of 0b001. Therefore, they are considered cacheable and read (burst) from memory and placed in the cache.
3. The PTE in the selected PTEG is tested for a match with the virtual page number (VPN) of the access. The VPN is the VSID concatenated with the page index field of the virtual address. For a match to occur, the following must be true:
   — PTE[H] = 0
   — PTE[V] = 1
   — PTE[VSID] = VA[0–23]
   — PTE[API] = VA[24–29]
4. If a match is not found, step 3 is repeated for each of the other seven PTEs in the primary PTEG. If a match is found, the table search process continues as described in step 8. If a match is not found within the 8 PTEs of the primary PTEG, the address of the secondary PTEG is generated.
5. The first PTE (PTE0) in the secondary PTEG is read from memory. Again, because PTE reads have a WIM bit combination of 0b001, an entire cache line is read into the on-chip cache.
6. The PTE in the selected secondary PTEG is tested for a match with the virtual page number (VPN) of the access. For a match to occur, the following must be true:
   — PTE[H] = 1
   — PTE[V] = 1
   — PTE[VSID] = VA[0–23]
   — PTE[API] = VA[24–29]
7. If a match is not found, step 6 is repeated for each of the other seven PTEs in the secondary PTEG. If it is never found, an exception is taken (step 9).
8. If a match is found, the PTE is written into the on-chip TLB and the R bit is updated in the PTE in memory (if necessary). If there is no memory protection violation, the C bit is also updated in memory (if the access is a write operation) and the table search is complete.
9. If a match is not found within the 8 PTEs of the secondary PTEG, the search fails, and a page fault exception condition occurs (either an ISI exception or a DSI exception).

Figure 5-9 and Figure 5-10 show how the conceptual model for the primary and secondary page table search operations, described in the *PowerPC Microprocessor Family: The Programming Environments* manual, are realized in Gekko.Figure 5-9 shows the case of a **dcbz** or **dcbz_l** instruction that is executed with W = 1 or I = 1, and that the R bit may be updated in memory (if required) before the operation is performed or the alignment exception occurs. The R bit may also be updated if memory protection is violated.

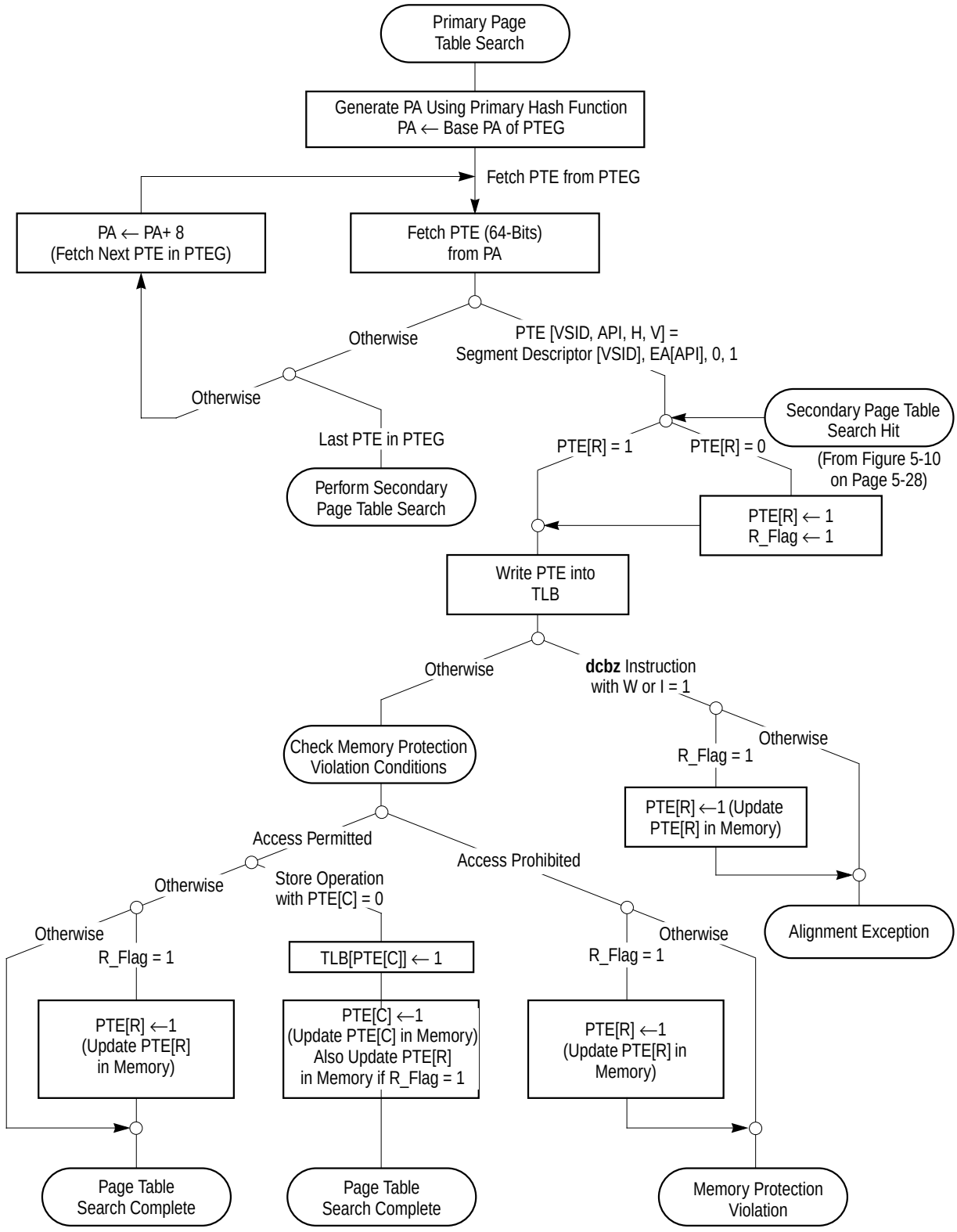


Figure 5-9. Primary Page Table Search

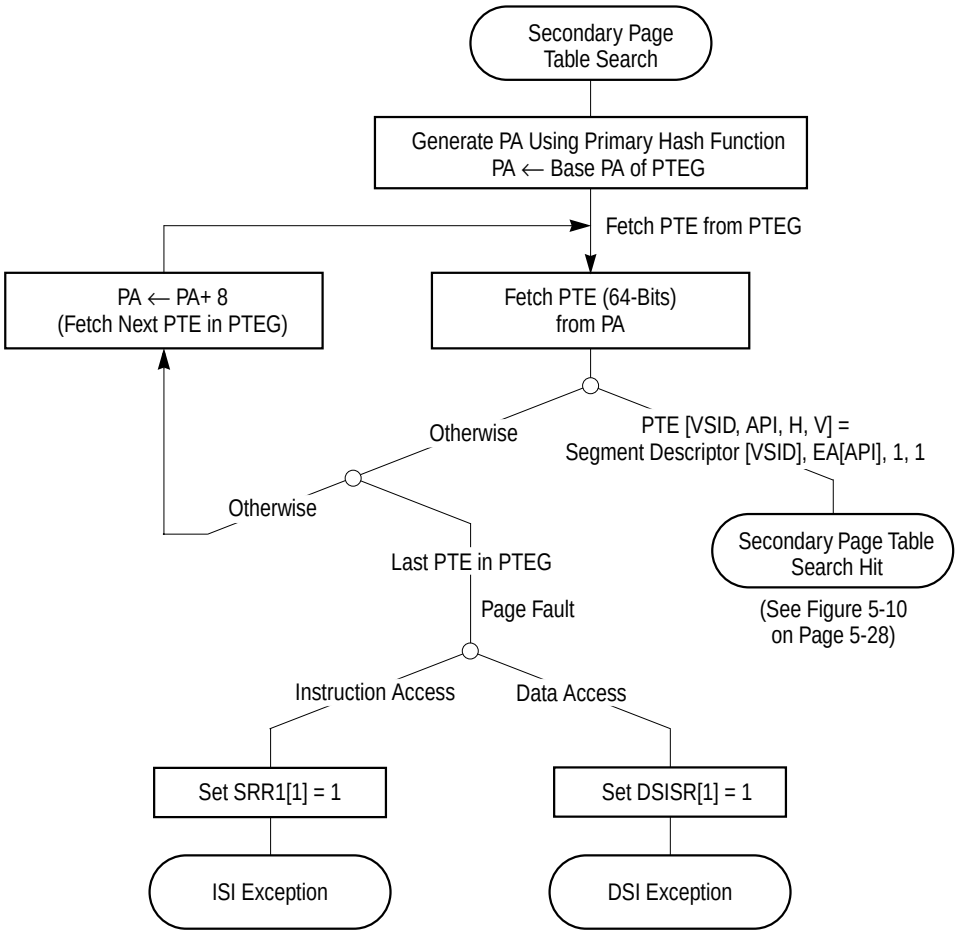


Figure 5-10. Secondary Page Table Search Flow

The LSU initiates out-of-order accesses without knowledge of whether it is legal to do so. Therefore, the MMU does not perform hardware table search due to TLB misses until the request is required by the program flow. In these out-of-order cases, the MMU does detect protection violations and whether a **dcbz** or **dcbz_l**instruction specifies a page marked as write-through or cache-inhibited. The MMU also detects alignment exceptions caused by the **dcbz** or **dcbz_l** instruction and prevents the changed bit in the PTE from being updated erroneously in these cases.

If an MMU register is being accessed by an instruction in the instruction stream, the IMMU stalls for one translation cycle to perform that operation. The sequencer serializes instructions to ensure the data correctness. For updating the IBATs and SRs, the sequencer classifies those operations as fetch serializing. After such an instruction is dispatched, the instruction buffer is flushed and the fetch stalls until the instruction completes. However, for reading from the IBATs, the operation is classified as execution serializing. As long as the LSU ensures that all previous instructions can be executed, subsequent instructions can be fetched and dispatched.

5.4.6 Page Table Updates

When TLBs are implemented (as in Gekko) they are defined as noncoherent caches of the page tables. TLB entries must be flushed explicitly with the TLB invalidate entry instruction (**tlbie**) whenever the corresponding PTE is modified. As Gekko is intended primarily for uniprocessor environments, it does not provide coherency of TLBs between multiple processors. If Gekko is used in a multiprocessor environment where TLB coherency is required, all synchronization must be implemented in software.

Processors may write referenced and changed bits with unsynchronized, atomic byte store operations. Note that the V, R, and C bits each reside in a distinct byte of a PTE. Therefore, extreme care must be taken to use byte writes when updating only one of these bits.

Explicitly altering certain MSR bits (using the **mtmsr** instruction), or explicitly altering PTEs, or certain system registers, may have the side effect of changing the effective or physical addresses from which the current instruction stream is being fetched. This kind of side effect is defined as an implicit branch. Implicit branches are not supported and an attempt to perform one causes boundedly-undefined results. Therefore, PTEs must not be changed in a manner that causes an implicit branch.

Chapter 2, “PowerPC Register Set” in the *PowerPC Microprocessor Family: The Programming Environments* manual, lists the possible implicit branch conditions that can occur when system registers and MSR bits are changed.

5.4.7 Segment Register Updates

Synchronization requirements for using the move to segment register instructions are described in “Synchronization Requirements for Special Registers and for Lookaside Buffers” in Chapter 2, “PowerPC Register Set” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Chapter 6 Instruction Timing

This chapter describes how the PowerPC Gekko microprocessor fetches, dispatches, and executes instructions and how it reports the results of instruction execution. It gives detailed descriptions of how Gekko's execution units work, and how those units interact with other parts of the processor, such as the instruction fetching mechanism, register files, and caches. It gives examples of instruction sequences, showing potential bottlenecks and how to minimize their effects. Finally, it includes tables that identify the unit that executes each instruction implemented on Gekko, the latency for each instruction, and other information that is useful for the assembly language programmer.

6.1 Terminology and Conventions

This section provides an alphabetical glossary of terms used in this chapter. These definitions are provided as a review of commonly used terms and as a way to point out specific ways these terms are used in this chapter.

- Branch prediction—The process of guessing whether a branch will be taken. Such predictions can be correct or incorrect; the term 'predicted' as it is used here does not imply that the prediction is correct (successful). The PowerPC architecture defines a means for static branch prediction as part of the instruction encoding.
- Branch resolution—The determination of whether a branch is taken or not taken. A branch is said to be resolved when the processor can determine which instruction path to take. If the branch is resolved as predicted, the instructions following the predicted branch that may have been speculatively executed can complete (see completion). If the branch is not resolved as predicted, instructions on the mispredicted path, and any results of speculative execution, are purged from the pipeline and fetching continues from the nonpredicted path.
- Completion—Completion occurs when an instruction has finished executing, written back any results, and is removed from the completion queue. When an instruction completes, it is guaranteed that this instruction and all previous instructions can cause no exceptions.
- Fall-through (branch fall-through)—A not-taken branch. On Gekko, fall-through branch instructions are removed from the instruction stream at dispatch. That is, these instructions are allowed to fall through the instruction queue via the dispatch mechanism, without either being passed to an execution unit and or given a position in the completion queue.
- Fetch—The process of bringing instructions from memory (such as a cache or system memory) into the instruction queue.
- Folding (branch folding)—The replacement with target instructions of a branch instruction and any instructions along the not-taken path when a branch is either taken or predicted as taken.
- Finish—Finishing occurs in the last cycle of execution. In this cycle, the completion queue entry is updated to indicate that the instruction has finished executing.
- Latency— The number of clock cycles necessary to execute an instruction and make ready the results of that execution for a subsequent instruction.
- Pipeline—In the context of instruction timing, the term 'pipeline' refers to the interconnection of the stages. The events necessary to process an instruction are broken into several cycle-length tasks to allow work to be performed on several instructions

simultaneously—analogous to an assembly line. As an instruction is processed, it passes from one stage to the next. When it does, the stage becomes available for the next instruction.

Although an individual instruction may take many cycles to complete (the number of cycles is called instruction latency), pipelining makes it possible to overlap the processing so that the throughput (number of instructions completed per cycle) is greater than if pipelining were not implemented.

- Program order—The order of instructions in an executing program. More specifically, this term is used to refer to the original order in which program instructions are fetched into the instruction queue from the cache.
- Rename register—Temporary buffers used by instructions that have finished execution but have not completed.
- Reservation station—A buffer between the dispatch and execute stages that allows instructions to be dispatched even though the results of instructions on which the dispatched instruction may depend are not available.
- Retirement—Removal of the completed instruction from the completion queue.
- Stage—The term ‘stage’ is used in two different senses, depending on whether the pipeline is being discussed as a physical entity or a sequence of events. In the latter case, a stage is an element in the pipeline during which certain actions are performed, such as decoding the instruction, performing an arithmetic operation, or writing back the results. A stage is typically described as taking a processor clock cycle to perform its operation; however, some events (such as dispatch and write-back) happen instantaneously, and may be thought to occur at the end of the stage.

  An instruction can spend multiple cycles in one stage. An integer multiply, for example, takes multiple cycles in the execute stage. When this occurs, subsequent instructions may stall.

  In some cases, an instruction may also occupy more than one stage simultaneously, especially in the sense that a stage can be seen as a physical resource—for example, when instructions are dispatched they are assigned a place in the completion queue at the same time they are passed to the execute stage. They can be said to occupy both the complete and execute stages in the same clock cycle.
- Stall—An occurrence when an instruction cannot proceed to the next stage.
- Superscalar—A superscalar processor is one that can issue multiple instructions concurrently from a conventional linear instruction stream. In a superscalar implementation, multiple instructions can be in the execute stage at the same time.
- Throughput—A measure of the number of instructions that are processed per cycle. For example, a series of double-precision floating-point multiply instructions has a throughput of one instruction per clock cycle.
- Write-back—Write-back (in the context of instruction handling) occurs when a result is written into the architectural registers (typically the GPRs and FPRs). Results are written back at completion time. Results in the write-back buffer cannot be flushed. If an exception occurs, these buffers must write back before the exception is taken.

6.2 Instruction Timing Overview

Gekko design minimizes average instruction execution latency, the number of clock cycles it takes to fetch, decode, dispatch, and execute instructions and make the results available for a subsequent instruction. Some instructions, such as loads and stores, access memory and require additional clock cycles between the execute phase and the write-back phase. These latencies vary depending on whether the access is to cacheable or noncacheable memory, whether it hits in the L1 or L2 cache, whether the cache access generates a write-back to memory, whether the access causes a snoop hit from another device that generates additional activity, and other conditions that affect memory accesses.

Gekko implements many features to improve throughput, such as pipelining, superscalar instruction issue, branch folding, removal of fall-through branches, two-level speculative branch handling, and multiple execution units that operate independently and in parallel.

As an instruction passes from stage to stage in a pipelined system, the following instruction can follow through the stages as the former instruction vacates them, allowing several instructions to be processed simultaneously. While it may take several cycles for an instruction to pass through all the stages, when the pipeline has been filled, one instruction can complete its work on every clock cycle.

The entire path that instructions take through the fetch, decode/dispatch, execute, complete, and write-back stages is considered Gekko's master pipeline, and two of the Gekko's execution units (the FPU and LSU) are also multiple-stage pipelines.

Gekko contains the following execution units that operate independently and in parallel:

- Branch processing unit (BPU)
- Integer unit 1 (IU1)—executes all integer instructions
- Integer unit 2 (IU2)—executes all integer instructions except multiplies and divides
- 64-bit floating-point unit (FPU)
- Load/store unit (LSU)
- System register unit (SRU)

Figure 6-1 represents a generic pipelined execution unit.

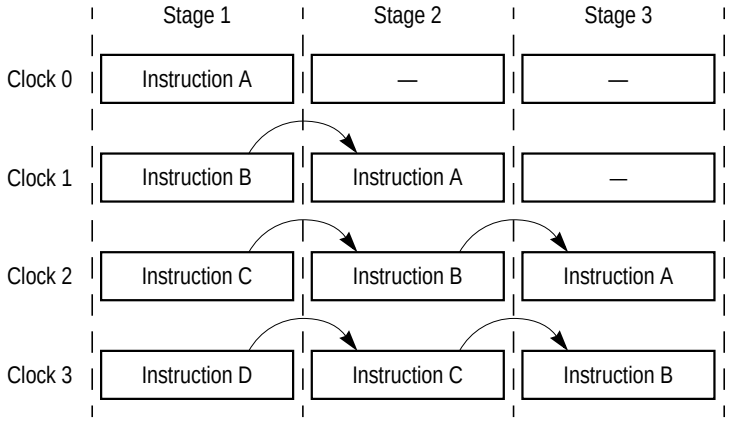


Figure 6-1. Pipelined Execution Unit

Gekko can retire two instructions on every clock cycle. In general, the Gekko processes instructions in four stages—fetch, decode/dispatch, execute, and complete as shown in Figure 6-2.

Note that the example of a pipelined execution unit in Figure 6-1 is similar to the three-stage FPU pipeline in Figure 6-2.

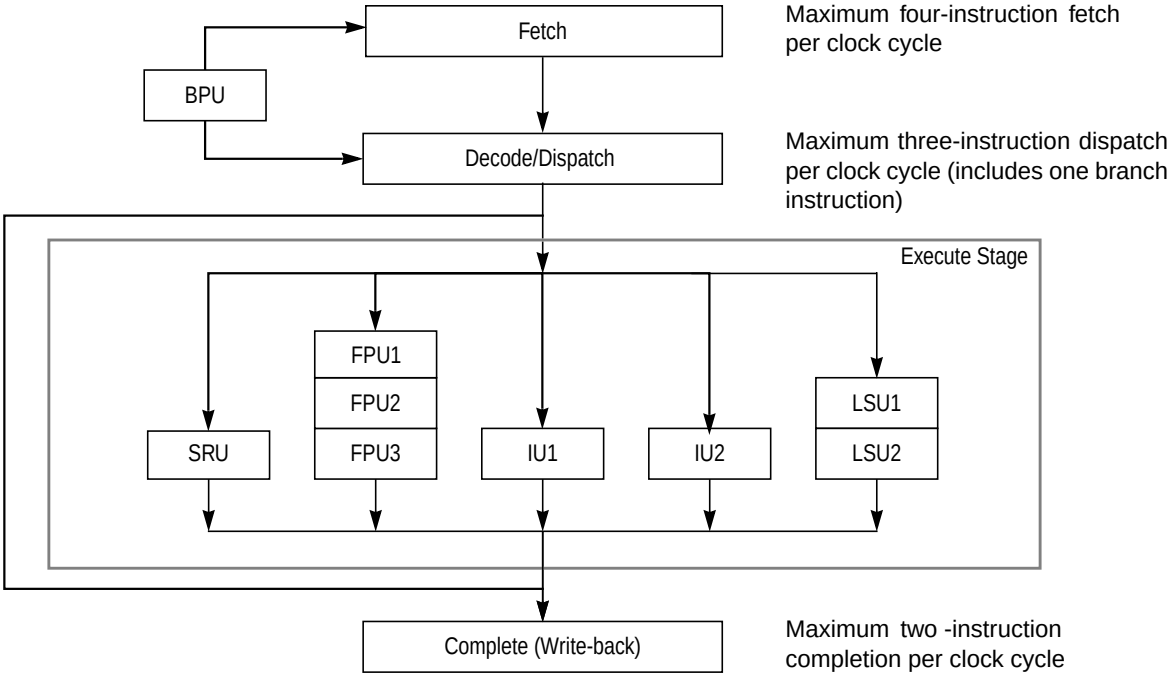


Figure 6-2. Superscalar/Pipeline Diagram

The instruction pipeline stages are described as follows:

- The instruction fetch stage includes the clock cycles necessary to request instructions from the memory system and the time the memory system takes to respond to the request. Instruction fetch timing depends on many variables, such as whether the instruction is in the branch target instruction cache, the on-chip instruction cache, or the L2 cache. Those factors increase when it is necessary to fetch instructions from system memory, and include the processor-to-bus clock ratio, the amount of bus traffic, and whether any cache coherency operations are required.

  Because there are so many variables, unless otherwise specified, the instruction timing examples below assume optimal performance, that the instructions are available in the instruction queue in the same clock cycle that they are requested. The fetch stage ends when the instruction is dispatched.

- The decode/dispatch stage consists of the time it takes to fully decode the instruction and dispatch it from the instruction queue to the appropriate execution unit. Instruction dispatch requires the following:
  - — Instructions can be dispatched only from the two lowest instruction queue entries, IQ0 and IQ1.
  - — A maximum of two instructions can be dispatched per clock cycle (although an additional branch instruction can be handled by the BPU).
  - — Only one instruction can be dispatched to each execution unit per clock cycle.
  - — There must be a vacancy in the specified execution unit.

  - A rename register must be available for each destination operand specified by the instruction.
  - For an instruction to dispatch, the appropriate execution unit must be available and there must be an open position in the completion queue. If no entry is available, the instruction remains in the IQ.
- The execute stage consists of the time between dispatch to the execution unit (or reservation station) and the point at which the instruction vacates the execution unit.

  Most integer instructions have a one-cycle latency; results of these instructions can be used in the clock cycle after an instruction enters the execution unit. However, integer multiply and divide instructions take multiple clock cycles to complete. The IU1 can process all integer instructions; the IU2 can process all integer instructions except multiply and divide instructions.

  The LSU and FPU are pipelined (as shown in Figure 6-2).
- The complete (complete/write-back) pipeline stage maintains the correct architectural machine state and commits it to the architectural registers at the proper time. If the completion logic detects an instruction containing an exception status, all following instructions are cancelled, their execution results in rename registers are discarded, and the correct instruction stream is fetched.

  The complete stage ends when the instruction is retired. Two instructions can be retired per cycle. Instructions are retired only from the two lowest completion queue entries, CQ0 and CQ1.

The notation conventions used in the instruction timing examples are as follows:

Fetch—The fetch stage includes the time between when an instruction is requested and when it is brought into the instruction queue. This latency can be very variable, depending upon whether the instruction is in the BTIC, the on-chip cache, the L2 cache, or system memory (in which case latency can be affected by bus speed and traffic on the system bus, and address translation issues). Therefore, in the examples in this chapters, the fetch stage is usually idealized, that is, an instruction is usually shown to be in the fetch stage when it is a valid instruction in the instruction queue. The instruction queue has six entries, IQ0–IQ5.

In dispatch entry (IQ0/IQ1)—Instructions can be dispatched from IQ0 and IQ1. Because dispatch is instantaneous, it is perhaps more useful to describe it as an event that marks the point in time between the last cycle in the fetch stage and the first cycle in the execute stage.

Execute—The operations specified by an instruction are being performed by the appropriate execution unit. The black stripe is a reminder that the instruction occupies an entry in the completion queue, described in Figure 6-3.

Complete—The instruction is in the completion queue. In the final stage, the results of the executed instruction are written back and the instruction is retired. The completion queue has six entries, CQ0–CQ5.

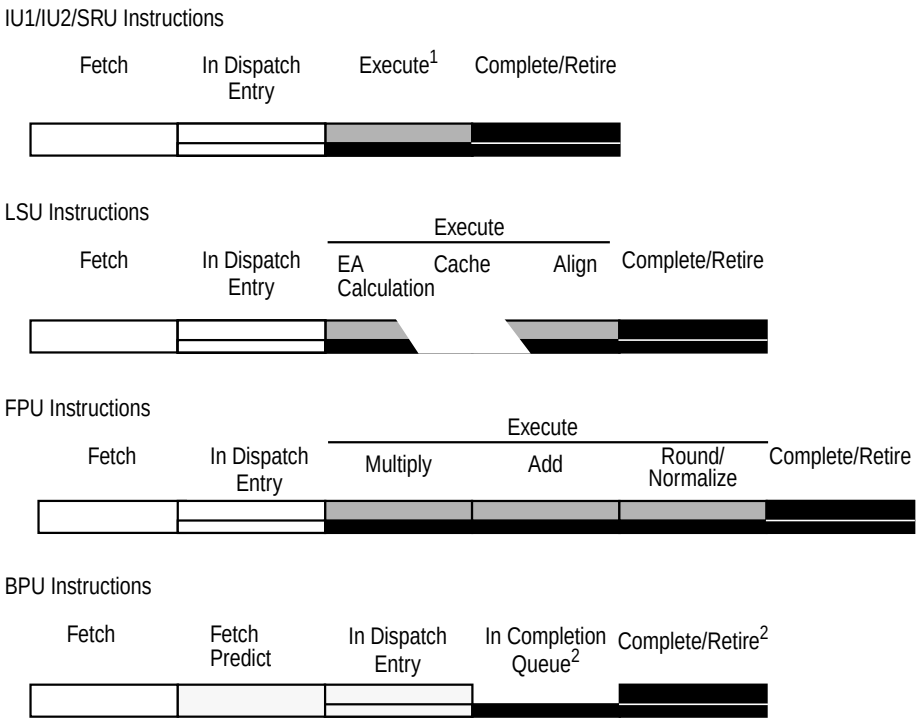
In retirement entry—Completed instructions can be retired from CQ0 and CQ1. Like dispatch, retirement is an event that in this case occurs at the end of the final cycle of the complete stage.

Figure 6-3 shows the stages of Gekko’s execution units.

6.3 Timing Considerations

Gekko is a superscalar processor; as many as three instructions can be issued to the execution units (one branch instruction to the branch processing unit, and two instructions issued from the dispatch queue to the other execution units) during each clock cycle. Only one instruction can be dispatched to each execution unit.

Although instructions appear to the programmer to execute in program order, Gekko improves performance by executing multiple instructions at a time, using hardware to manage dependencies. When an instruction is dispatched, the register file provides the source data to the execution unit. The register files and rename register have sufficient bandwidth to allow dispatch of two instructions per clock under most conditions.



1 Several integer instructions, such as multiply and divide instructions, require multiple cycles in the execute stage.

2 Only those branch instructions that update the LR or CTR take an entry in the completion queue.

Figure 6-3. PowerPC Gekko Microprocessor Pipeline Stages

Gekko's BPU decodes and executes branches immediately after they are fetched. When a conditional branch cannot be resolved due to a CR data dependency, the branch direction is predicted and execution continues from the predicted path. If the prediction is incorrect, the following steps are taken:

1. The instruction queue is purged and fetching continues from the correct path.
2. Any instructions ahead of the predicted branch in the completion queue are allowed to complete.
3. Instructions after the mispredicted branch are purged.
4. Dispatching resumes from the correct path.

After an execution unit finishes executing an instruction, it places resulting data into the appropriate GPR or FPR rename register. The results are then stored into the correct GPR or FPR during the write-back stage. If a subsequent instruction needs the result as a source operand, it is made available simultaneously to the appropriate execution unit, which allows a data-dependent instruction to be decoded and dispatched without waiting to read the data from the register file. Branch instructions that update either the LR or CTR write back their results in a similar fashion.

The following section describes this process in greater detail.

6.3.1 General Instruction Flow

As many as four instructions can be fetched into the instruction queue (IQ) in a single clock cycle. Instructions enter the IQ and are issued to the various execution units from the dispatch queue. Gekko tries to keep the IQ full at all times, unless instruction cache throttling is operating.

The number of instructions requested in a clock cycle is determined by the number of vacant spaces in the IQ during the previous clock cycle. This is shown in the examples in this chapter. Although the instruction queue can accept as many as four new instructions in a single clock cycle, if only one IQ entry is vacant, only one instruction is fetched. Typically instructions are fetched from the on-chip instruction cache, but they may also be fetched from the branch target instruction cache (BTIC). If the instruction request hits in the BTIC, it can usually present the first two instructions of the new instruction stream in the next clock cycle, giving enough time for the next pair of instructions to be fetched from the instruction cache with no idle cycles. If instructions are not in the BTIC or the on-chip instruction cache, they are fetched from the L2 cache or from system memory.

Gekko's instruction cache throttling feature, managed through the instruction cache throttling control (ICTC) register, can lower the processor's overall junction temperature by slowing the instruction fetch rate. See Chapter 10, "Power and Thermal Management" for more information.

Branch instructions are identified by the fetcher, and forwarded to the BPU directly, bypassing the dispatch queue. If the branch is unconditional or if the specified conditions are already known, the branch can be resolved immediately. That is, the branch direction is known and instruction fetching can continue from the correct location. Otherwise, the branch direction must be predicted. Gekko offers several resources to aid in quick resolution of branch instructions and for improving the accuracy of branch predictions. These include the following:

- Branch target instruction cache—The 64-entry (four-way-associative) branch target instruction cache (BTIC) holds branch target instructions so when a branch is encountered in a repeated loop, usually the first two instructions in the target stream can be fetched into the instruction queue on the next clock cycle. The BTIC can be disabled and invalidated through bits in HID0.

- Dynamic branch prediction—The 512-entry branch history table (BHT) is implemented with two bits per entry for four degrees of prediction—not-taken, strongly not-taken, taken, strongly taken. Whether a branch instruction is taken or not-taken can change the strength of the next prediction. This dynamic branch prediction is not defined by the PowerPC architecture.

  To reduce aliasing, only predicted branches update the BHT entries. Dynamic branch prediction is enabled by setting HID0[BHT]; otherwise, static branch prediction is used.

- Static branch prediction—Static branch prediction is defined by the PowerPC architecture and involves encoding the branch instructions. See 6.4.1.3.1."

Branch instructions that do not update the LR or CTR are removed from the instruction stream either by branch folding or removal of fall-through branch instructions, as described in 6.4.1.1." Branch instructions that update the LR or CTR are treated as if they require dispatch (even through they are not issued to an execution unit in the process). They are assigned a position in the completion queue to ensure that the CTR and LR are updated sequentially.

All other instructions are issued from the IQ0 and IQ1. The dispatch rate depends upon the availability of resources such as the execution units, rename registers, and completion queue entries, and upon the serializing behavior of some instructions. Instructions are dispatched in program order; an instruction in IQ1 cannot be dispatched ahead of one in IQ0.

6.3.2 Instruction Fetch Timing

Instruction fetch latency depends on whether the fetch hits the BTIC, the on-chip instruction cache, or the L2 cache. If no cache hit occurs, a memory transaction is required in which case fetch latency is affected by bus traffic, bus clock speed, and memory translation. These issues are discussed further in the following sections.

6.3.2.1 Cache Arbitration

When the instruction fetcher requests instructions from the instruction cache, two things may happen. If the instruction cache is idle and the requested instructions are present, they are provided on the next clock cycle. However, if the instruction cache is busy due to a cache-line-reload operation, instructions cannot be fetched until that operation completes.

6.3.2.2 Cache Hit

If the instruction fetch hits the instruction cache, it takes only one clock cycle after the request for as many as four instructions to enter the instruction queue. Note that the cache is not blocked to internal accesses during a cache reload completes (hits under misses). The critical double word is written simultaneously to the cache and forwarded to the requesting unit, minimizing stalls due to load delays.

Figure 6-4 shows the paths taken by instructions.

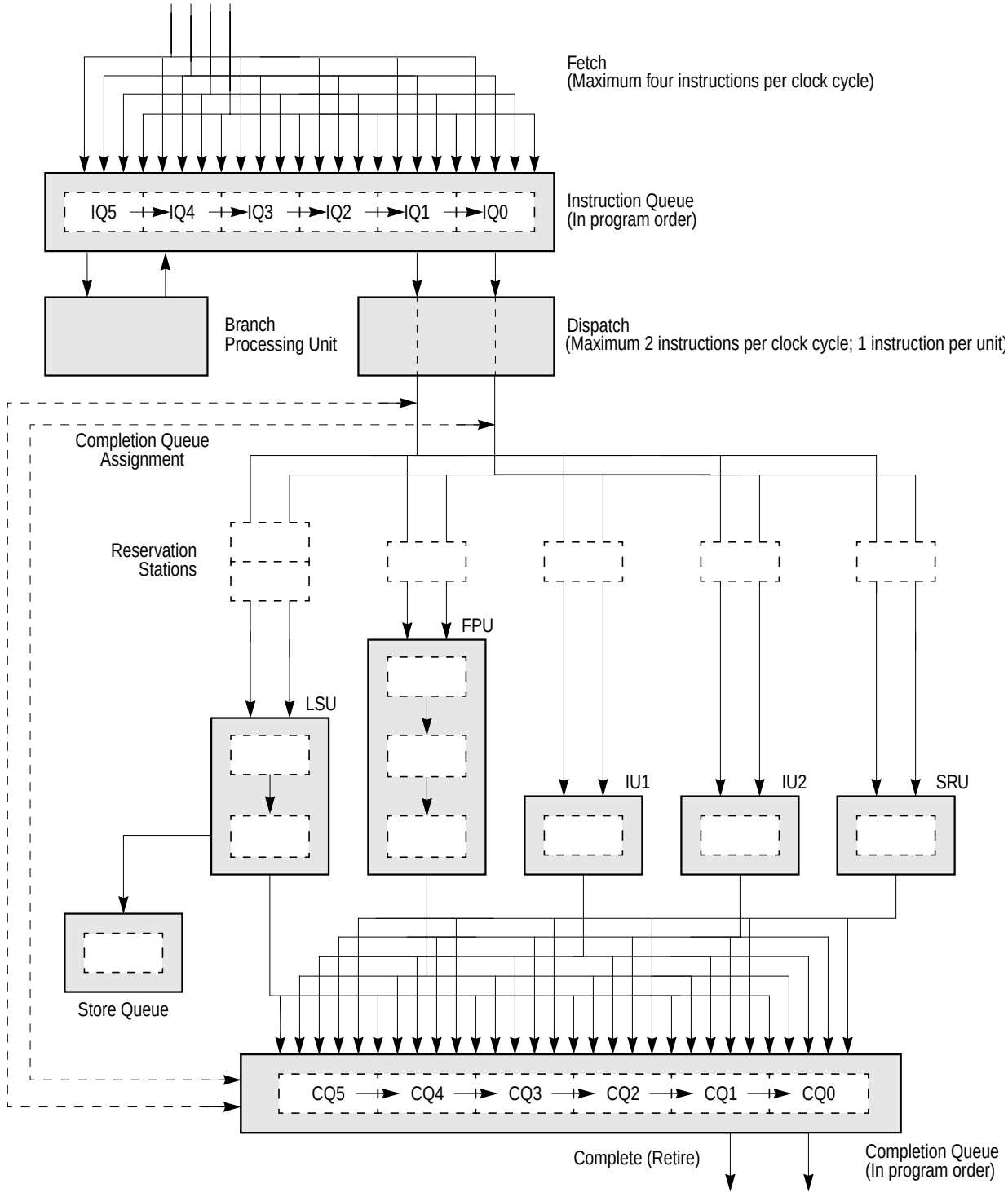


Figure 6-4. Instruction Flow Diagram

Figure 6-5 shows a simple example of instruction fetching that hits in the on-chip cache. This example uses a series of integer add and double-precision floating-point add instructions to show how the number of instructions to be fetched is determined, how program order is maintained by the instruction and completion queues, how instructions are dispatched and retired in pairs (maximum), and how the FPU, IU1, and IU2 pipelines function. The following instruction sequence is examined:

```
0    add
1    fadd
2    add
3    fadd
4    br 6
5    fsub
6    fadd
7    fadd
8    add
9    add
10   add
11   add
12   fadd
13   add
14   fadd
15   .
16   .
17   .
```

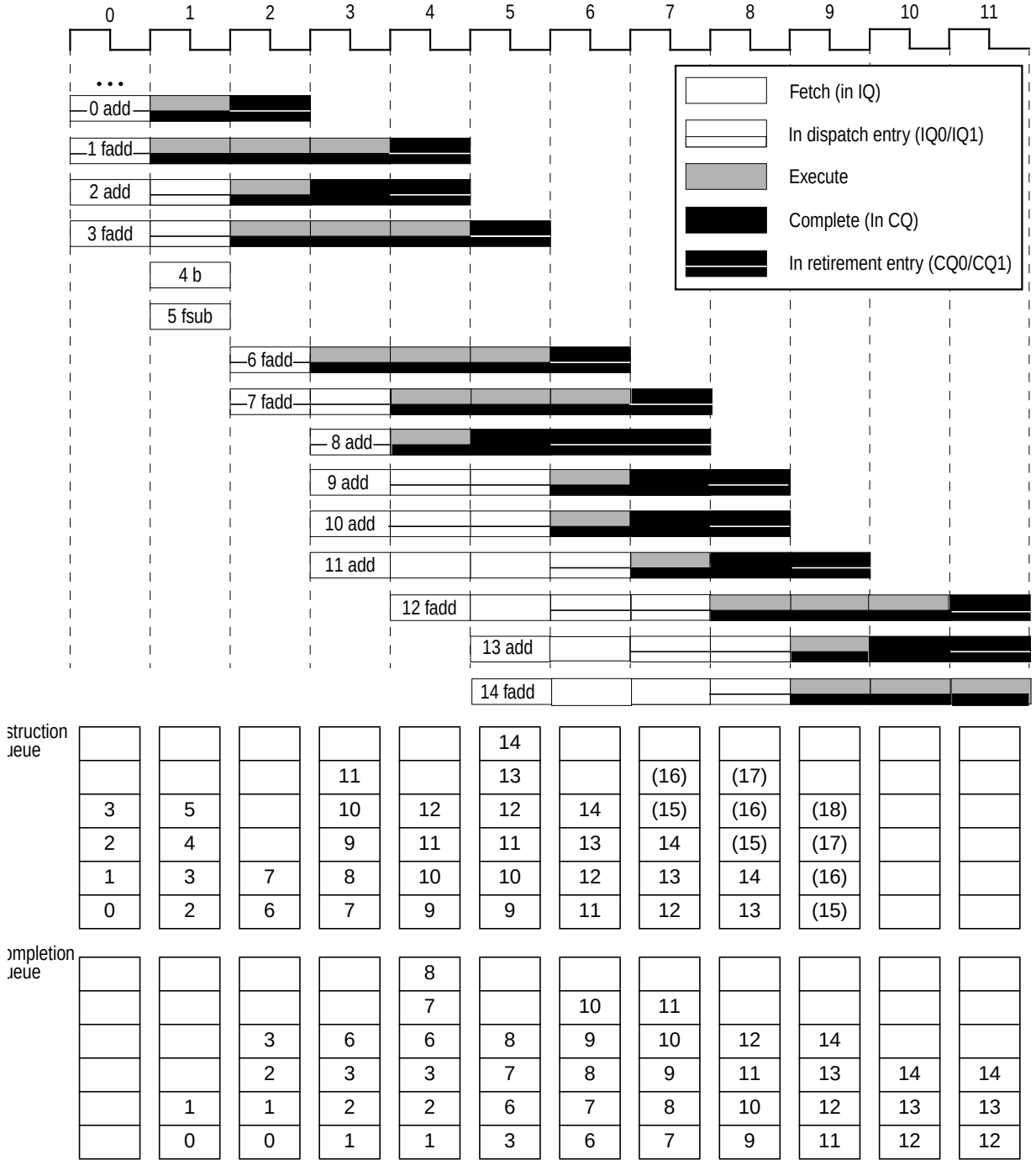


Figure 6-5. Instruction Timing—Cache Hit

The instruction timing for this example is described cycle-by-cycle as follows:

0. In cycle 0, instructions 0–3 are fetched from the instruction cache. Instructions 0 and 1 are placed in the two entries in the instruction queue from which they can be dispatched on the next clock cycle.

1. In cycle 1, instructions 0 and 1 are dispatched to the IU2 and FPU, respectively. Notice that for instructions to be dispatched they must be assigned positions in the completion queue. In this case, since the completion queue was empty, instructions 0 and 1 take the two lowest entries in the completion queue. Instructions 2 and 3 drop into the two dispatch positions in the instruction queue. Because there were two positions available in the instruction queue in clock cycle 0, two instructions (4 and 5) are fetched into the instruction queue. Instruction 4 is a branch unconditional instruction, which resolves immediately as taken. Because the branch is taken, it can therefore be folded from the instruction queue.
2. In cycle 2, assume a BTIC hit occurs and target instructions 6 and 7 are fetched into the instruction queue, replacing the folded **b** instruction (4) and instruction 5. Instruction 0 completes, writes back its results and vacates the completion queue by the end of the clock cycle. Instruction 1 enters the second FPU execute stage, instruction 2 is dispatched to the IU2, and instruction 3 is dispatched into the first FPU execute stage. Because the taken branch instruction (4) does not update either CTR or LR, it does not require a position in the completion queue and can be folded.
3. In cycle 3, target instructions (6 and 7) are fetched, replacing instructions 4 and 5 in IQ0 and IQ1. This replacement on taken branches is called branch folding. Instruction 1 proceeds through the last of the three FPU execute stages. Instruction 2 has executed but must remain in the completion queue until instruction 1 completes. Instruction 3 replaces instruction 1 in the second stage of the FPU, and instruction 6 replaces instruction 3 in the first stage.

   Because there were four vacancies in the instruction queue in the previous clock cycle, instructions 8–11 are fetched in this clock cycle.
4. Instruction 1 completes in cycle 4, allowing instruction 2 to complete. Instructions 3 and 6 continue through the FPU pipeline. Because there were two openings in the completion queue in the previous cycle, instructions 7 and 8 are dispatched to the FPU and IU2, respectively, filling the completion queue. Similarly, because there was one opening in the instruction queue in clock cycle 3, one instruction is fetched.
5. In cycle 5, instruction 3 completes, and instructions 13 and 14 are fetched. Instructions 6 and 7 continue through the FPU pipeline. No instructions are dispatched in this clock cycle because there were no vacant CQ entries in cycle 4.
6. In cycle 6, instruction 6 completes, instruction 7 is in stage 3 of the FPU execute stage, and although instruction 8 has executed, it must wait for instruction 7 to complete. The two integer instructions, 9 and 10, are dispatched to the IU2 and IU1, respectively. No instructions are fetched because the instruction queue was full on the previous cycle.
7. In cycle 7, instruction 7 completes, allowing instruction 8 to complete as well. Instructions 9 and 10 remain in the completion stage, since at most two instructions can complete in a cycle. Because there was one opening in the completion queue in cycle 6, instructions 11 is dispatched to the IU2. Two more instructions (15 and 16, which are shown only in the instruction queue) are fetched.
8. In cycle 8, instructions 9–11 are through executing. Instructions 9 and 10 complete, write back, and vacate the completion queue. Instruction 11 must wait to complete on the following cycle. Because the completion queue had one opening in the previous cycle, instruction 12 can be dispatched to the FPU. Similarly, the instruction queue had one opening in the previous cycle, so one additional instruction, 17, can be fetched.

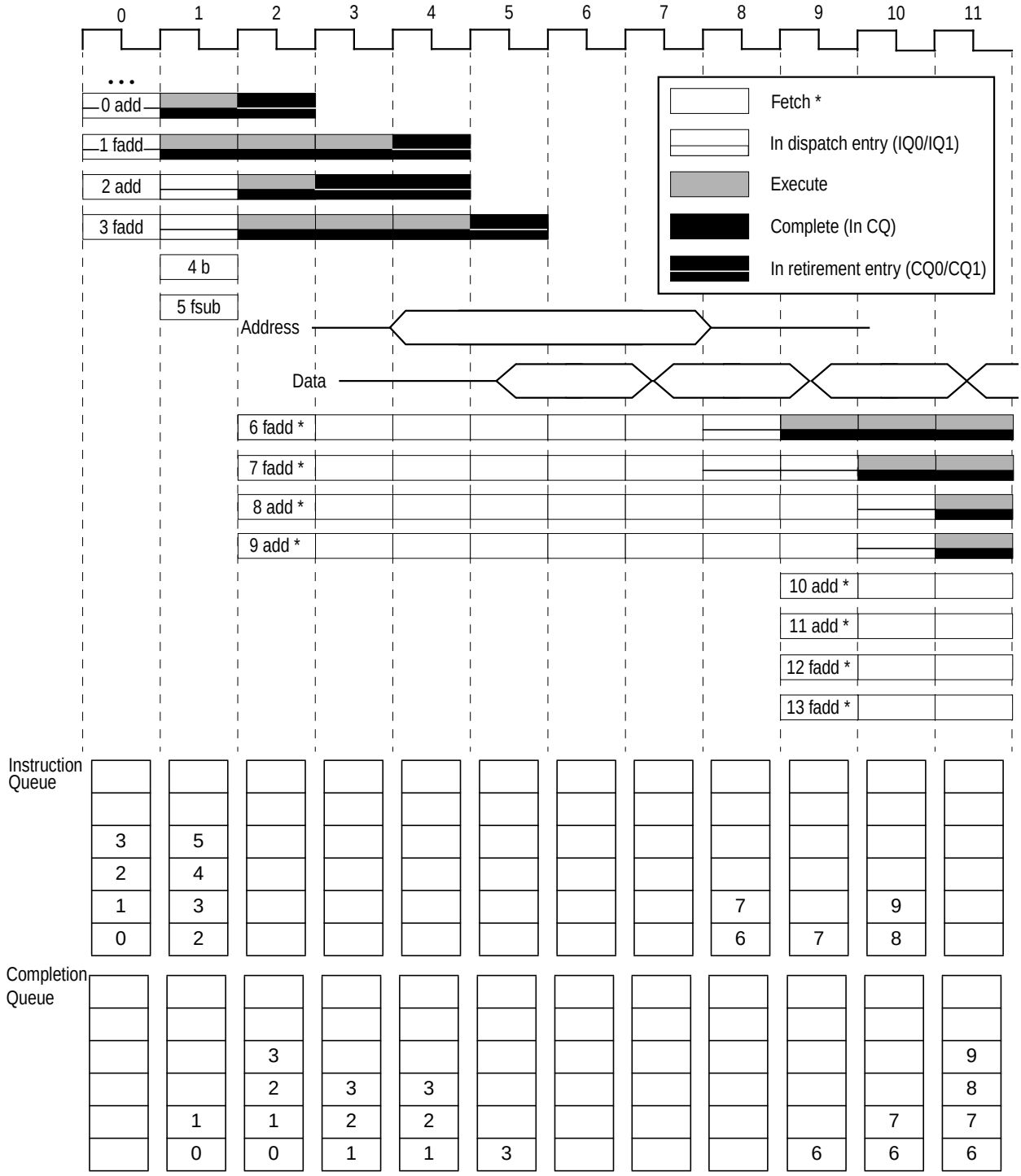
9. In cycle 9, instruction 11 completes, instruction 12 continues through the FPU pipeline, and instructions 13 and 14 are dispatched. One new instruction, 18, can be fetched on this cycle because the instruction queue had one opening on the previous clock cycle.

6.3.2.3 Cache Miss

Figure 6-6 shows an instruction fetch that misses both the on-chip cache and L2 cache. A processor/bus clock ratio is 1:2 is used. The same instruction sequence is used as in 6.3.2.2," however in this example, the branch target instruction is not in either the L1 or L2 cache. Because the target instruction is not in the L1 cache, it cannot be in the BTIC.

A cache miss, extends the latency of the fetch stage, so in this example, the fetch stage shown represents not only the time the instruction spends in the IQ, but the time required for the instruction to be loaded from system memory, beginning in clock cycle 2.

During clock cycle 3, the target instruction for the **b** instruction is not in the BTIC, the instruction cache or the L2 cache; therefore, a memory access must occur. During clock cycle 5, the address of the block of instructions is sent to the system bus. During clock cycle 7, two instructions (64 bits) are returned from memory on the first beat and are forwarded both to the cache and the instruction fetcher.



* Instructions 5 and 6 are not in the IQ in clock cycle 5. Here, the fetch stage shows cache latency.

Figure 6-6. Instruction Timing—Cache Miss

6.3.2.4 L2 Cache Access Timing Considerations

If an instruction fetch misses both the BTIC and the on-chip instruction cache, the Gekko next looks in the L2 cache. If the requested instructions are there, they are burst into the Gekko in much the same way as shown in Figure 6-6. The formula for the L2 cache latency for instruction accesses is as follows:

1 processor clock + 3 L2 clocks + 1 processor clock

Therefore, since the L2 is operating in 1:1 mode, the instruction fetch takes 5 processor clock cycles.

6.3.2.5 Instruction Dispatch and Completion Considerations

Several factors affect Gekko's ability to dispatch instructions at a peak rate of two per cycle—the availability of the execution unit, destination rename registers, and completion queue, as well as the handling of completion-serialized instructions. Several of these limiting factors are illustrated in the previous instruction timing examples.

To reduce dispatch unit stalls due to instruction data dependencies, Gekko provides a single-entry reservation station for the FPU, SRU, and each IU, and a two-entry reservation station for the LSU. If a data dependency keeps an instruction from starting execution, that instruction is dispatched to the reservation station associated with its execution unit (and the rename registers are assigned), thereby freeing the positions in the instruction queue so instructions can be dispatched to other execution units. Execution begins during the same clock cycle that the rename buffer is updated with the data the instruction is dependent on.

If both instructions in IQ0 and IQ1 require the same execution unit, the instruction in IQ1 cannot be dispatched until the first instruction proceeds through the pipeline and provides the subsequent instruction with a vacancy in the requested execution unit.

The completion unit maintains program order after instructions are dispatched from the instruction queue, guaranteeing in-order completion and a precise exception model. Completing an instruction implies committing execution results to the architected destination registers. In-order completion ensures the correct architectural state when Gekko must recover from a mispredicted branch or an exception.

Instruction state and all information required for completion is kept in the six-entry, first-in/first-out completion queue. An completion queue entry is allocated for each instruction when it is dispatched to an execute unit; if no entry is available, the dispatch unit stalls. A maximum of two instructions per cycle may be completed and retired from the completion queue, and the flow of instructions can stall when a longer-latency instruction reaches the last position in the completion queue. Subsequent instructions cannot be completed and retired until that longer-latency instruction completes and retires. Examples of this are shown in 6.3.2.2," and 6.3.2.3."

Gekko can execute instructions out-of-order, but in-order completion by the completion unit ensures a precise exception mechanism. Program-related exceptions are signaled when the instruction causing the exception reaches the last position in the completion queue. Prior instructions are allowed to complete before the exception is taken.

6.3.2.6 Rename Register Operation

To avoid contention for a given register file location in the course of out-of-order execution, Gekko provides rename registers for holding instruction results before the completion commits them to the architected register. There are six GPR rename registers, six FPR rename registers, and one each for the CR, LR, and CTR.

When the dispatch unit dispatches an instruction to its execution unit, it allocates a rename register (or registers) for the results of that instruction. If an instruction is dispatched to a reservation station associated with an execution unit due to a data dependency, the dispatcher also provides a tag to the execution unit identifying the rename register that forwards the required data at completion. When the source data reaches the rename register, execution can begin.

Instruction results are transferred from the rename registers to the architected registers by the completion unit when an instruction is retired from the completion queue without exceptions and after any predicted branch conditions preceding it in the completion queue have been resolved correctly. If a branch prediction was incorrect, the instructions following the branch are flushed from the completion queue, and any results of those instructions are flushed from the rename registers.

6.3.2.7 Instruction Serialization

Although Gekko can dispatch and complete two instructions per cycle, so-called serializing instructions limit dispatch and completion to one instruction per cycle. There are three types of instruction serialization:

- Execution serialization—Execution-serialized instructions are dispatched, held in the functional unit and do not execute until all prior instructions have completed. A functional unit holding an execution-serialized instruction will not accept further instructions from the dispatcher. For example, execution serialization is used for instructions that modify nonrenamed resources. Results from these instructions are generally not available or forwarded to subsequent instructions until the instruction completes (using **mtspr** to write to LR or CTR does provide forwarding to branch instructions).
- Completion serialization (also referred to as post-dispatch or tail serialization)—Completion-serialized instructions inhibit dispatching of subsequent instructions until the serialized instruction completes. Completion serialization is used for instructions that bypass the normal rename mechanism.
- Refetch serialization (flush serialization)—Refetch-serialized instructions inhibit dispatch of subsequent instructions and force refetching of subsequent instructions after completion.

6.4 Execution Unit Timings

The following sections describe instruction timing considerations within each of the respective execution units in Gekko.

6.4.1 Branch Processing Unit Execution Timing

Flow control operations (conditional branches, unconditional branches, and traps) are typically expensive to execute in most machines because they disrupt normal flow in the instruction stream. When a change in program flow occurs, the IQ must be reloaded with the target instruction stream. Previously issued instructions will continue to execute while the new instruction stream makes its way into the IQ, but depending on whether the target instruction is in the BTIC, instruction cache, L2 cache, or in system memory, some opportunities may be missed to execute instructions, as the example in 6.3.2.3," shows.

Performance features such as the branch folding, removal of fall-through branch instructions, BTIC, dynamic branch prediction (implemented in the BHT), two-level branch prediction, and the implementation of nonblocking caches minimize the penalties associated with flow control operations on Gekko. The timing for branch instruction execution is determined by many factors including the following:

- Whether the branch is taken
- Whether instructions in the target stream, typically the first two instructions in the target stream, are in the branch target instruction cache (BTIC)
- Whether the target instruction stream is in the on-chip cache
- Whether the branch is predicted
- Whether the prediction is correct

6.4.1.1 Branch Folding and Removal of Fall-Through Branch Instructions

When a branch instruction is encountered by the fetcher, the BPU immediately begins to decode it and tries to resolve it. All branch instructions except those that update either the LR or CTR are removed from the instruction flow before they would take a position in the completion queue.

Branch folding occurs either when a branch is taken or is predicted as taken (as is the case with unconditional branches). When the BPU folds the branch instruction out of the instruction stream, the target instruction stream that is fetched into the instruction queue overwrites the branch instruction.

Figure 6-7 shows branch folding. Here a **br** instruction is encountered in a series of **add** instructions. The branch is resolved as taken. What happens on the next clock cycle depends on whether the target instruction stream is in the BTIC, the instruction cache, or if it must be fetched from the L2 cache or from system memory.

Figure 6-7 shows cases where there is a BTIC hit, and when there is a BTIC miss (and instruction cache hit).

If there is a BTIC hit on the next clock cycle the **b** instruction is replaced by the target instruction, **and**1, that was found in the BTIC; the second **and** instruction is also fetched from the BTIC. On the next clock cycle, the next four **and** instructions from the target stream are fetched from the instruction cache.

If the target instruction is not in the BTIC, there is an idle cycle while the fetcher attempts to fetch the first four instructions from the instruction cache (on the next clock cycle). In the example in Figure 6-7, the first four target instruction are fetched on the next clock.

If it misses in the caches, an L2 cache or memory access is required, the latency of which is

dependent on several factors, such as processor/bus clock ratios. In most cases, new instructions arrive in the IQ before the execution units become idle.

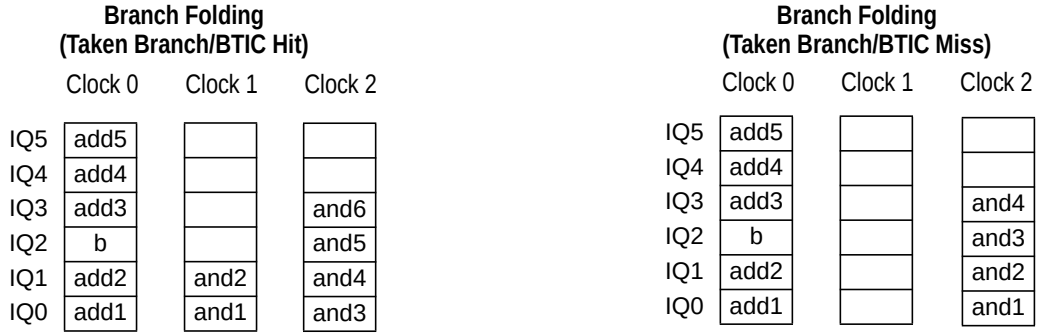


Figure 6-7. Branch Folding

Figure 6-8 shows the removal of fall-through branch instructions, which occurs when a branch is not taken or is predicted as not taken.

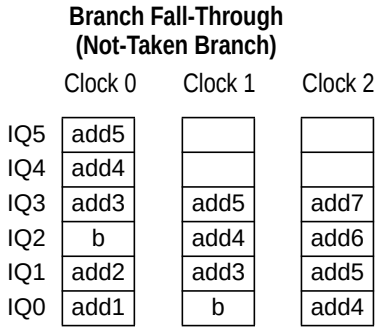


Figure 6-8. Removal of Fall-Through Branch Instruction

In this case the branch instruction remains in the instruction queue and is removed from the instruction stream as if it were dispatched. However, it is not dispatched to an execution unit and is not assigned an entry in the completion queue.

When a branch instruction is detected before it reaches a dispatch position, and if the branch is correctly predicted as taken, folding the branch instruction (and any instructions from the incorrect path) reduces the latency required for flow control to zero; instruction execution proceeds as though the branch was never there.

The advantage of removing the fall-through branch instructions at dispatch is only marginally less than that of branch folding. Because the branch is not taken, only the branch instruction needs to be discarded. The only cost of expelling the branch instruction from one of the dispatch entries rather than folding it is missing a chance to dispatch an executable instruction from that position.

6.4.1.2 Branch Instructions and Completion

As described in the previous section, instructions that do not update either the LR or CTR are removed from the instruction stream before they reach the completion queue, either by branch folding (in the case of taken branches) or by removing fall-through branch instructions at dispatch (in the case of non-taken branches). However, branch instructions that update the architected LR and CTR must do

so in program order and therefore must perform write-back in the completion stage, like the instructions that update the FPRs and GPRs.

Branch instructions that update the CTR or LR pass through the instruction queue like nonbranch instructions. At the point of dispatch, however, they are not sent to an execution unit, but rather are assigned a slot in the completion queue, as shown in Figure 6-9.

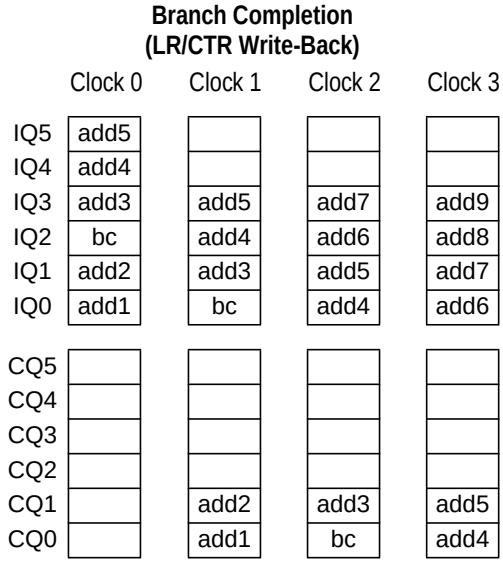


Figure 6-9. Branch Completion

In this example, the **bc** instruction is encoded to decrement the CTR. It is predicted as not-taken in clock cycle 0. In clock cycle 2, **bc** and **add**3 are both dispatched. In clock cycle 3, the architected CTR is updated and the **bc** instruction is retired from the completion queue.

6.4.1.3 Branch Prediction and Resolution

Gekko supports the following two types of branch prediction:

- Static branch prediction—This is defined by the PowerPC architecture as part of the encoding of branch instructions.
- Dynamic branch prediction—This is a processor-specific mechanism implemented in hardware (in particular the branch history table, or BHT) that monitors branch instruction behavior and maintains a record from which the next occurrence of the branch instruction is predicted.

When a conditional branch cannot be resolved due to a CR data dependency, the BPU predicts whether it will be taken, and instruction fetching proceeds down the predicted path. If the branch prediction resolves as incorrect, the instruction queue and all subsequently executed instructions are purged, instructions executed prior to the predicted branch are allowed to complete, and instruction fetching resumes down the correct path.

Gekko executes through two levels of prediction. Instructions from the first unresolved branch can execute, but they cannot complete until the branch is resolved. If a second branch instruction is encountered in the predicted instruction stream, it can be predicted and instructions can be fetched, but not executed, from the second branch. No action can be taken for a third branch instruction until at least one of the two previous branch instructions is resolved.

The number of instructions that can be executed after the issue of a predicted branch instruction is limited by the fact that no instruction executed after a predicted branch may actually update the register files or memory until the branch is completed. That is, instructions may be issued and executed, but cannot reach the write-back stage in the completion unit. When an instruction following a predicted branch completes execution, it does not write back its results to the architected registers, instead, it stalls in the completion queue. Of course, when the completion queue is full, no additional instructions can be dispatched, even if an execution unit is idle.

In the case of a misprediction, Gekko can easily redirect its machine state because the programming model has not been updated. When a branch is mispredicted, all instructions that were dispatched after the predicted branch instruction are flushed from the completion queue and any results are flushed from the rename registers.

The BTIC is a cache of recently used branch target instructions. If the search for the branch target hits in the cache, the first one or two branch instructions is available in the instruction queue on the next cycle (shown in Figure 6-5). Two instructions are fetched on a BTIC hit, unless the branch target is the last instruction in a cache block, in which case one instruction is fetched.

In some situations, an instruction sequence creates dependencies that keep a branch instruction from being resolved immediately, thereby delaying execution of the subsequent instruction stream based on the predicted outcome of the branch instruction. The instruction sequences and the resulting action of the branch instruction are described as follows:

- An **mtspr**(LK) followed by a **bclr**—Fetching stops and the branch waits for the **mtspr** to execute.
- An **mtspr**(CTR) followed by a **bcctr**—Fetching stops and the branch waits for the **mtspr** to execute.
- An **mtspr**(CTR) followed by a **bc** (CTR decrement)—Fetching stops and the branch waits for the **mtspr** to execute.
- A third **bc**(based-on-CR) is encountered while there are two unresolved **bc**(based-on-CR). The third **bc**(based-on-CR) is not executed and fetching stops until one of the previous **bc**(based-on-CR) is resolved. (Note that branch conditions can be a function of the CTR and the CR; if the CTR condition is sufficient to resolve the branch, then a CR-dependency is ignored.)

6.4.1.3.1 Static Branch Prediction

The PowerPC architecture provides a field in branch instructions (the BO field) to allow software to hint whether a branch is likely to be taken. Rather than delaying instruction processing until the condition is known, Gekko uses the instruction encoding to predict whether the branch is likely to be taken and begins fetching and executing along that path. When the branch condition is known, the prediction is evaluated. If the prediction was correct, program flow continues along that path; otherwise, the processor flushes any instructions and their results from the mispredicted path, and program flow resumes along the correct path.

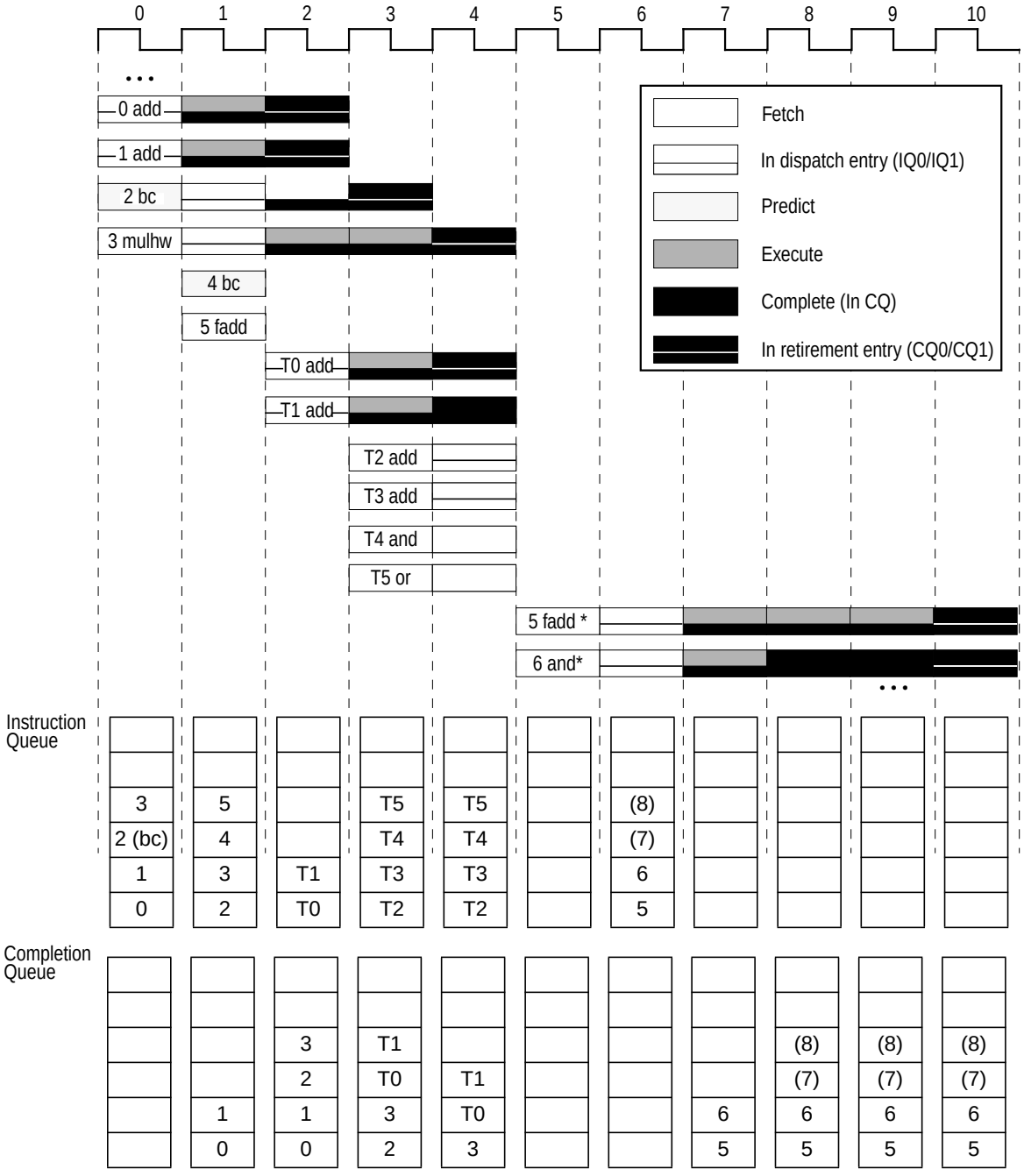
Static branch prediction is used when HID0[BHT] is cleared. That is, the branch history table, which is used for dynamic branch prediction, is disabled.

For information about static branch prediction, see “Conditional Branch Control,” in Chapter 4, “Addressing Modes and Instruction Set Summary” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

6.4.1.3.2 Predicted Branch Timing Examples

Figure 6-10 shows cases where branch instructions are predicted. It shows how both taken and not-taken branches are handled and how Gekko handles both correct and incorrect predictions. The example shows the timing for the following instruction sequence:

```
0    add
1    add
2    bc
3    mulhw
4    bc T0
5    fadd
6    and
add
T7   add
T8   add
T9   add
T10  add
T11  or
```



* Instructions 5 and 6 are not in the IQ in clock cycle 5. Here, the fetch stage shows cache latency.

Figure 6-10. Branch Instruction Timing

0. During clock cycle 0, instructions 0 and 1 are dispatched to their respective execution units. Instruction 2 is a branch instruction that updates the CTR. It is predicted as not taken in clock cycle 0. Instruction 3 is a **mulhw** instruction on which instruction 4 depends.
1. In clock cycle 1, instructions 2 and 3 enter the dispatch entries in the IQ. Instruction 4 (a second **bc** instruction) and 5 are fetched. The second **bc** instruction is predicted as taken. It can be folded, but it cannot be resolved until instruction 3 writes back.

2. In clock cycle 2, instruction 4 has been folded and instruction 5 has been flushed from the IQ. The two target instructions, T0 and T1, are both in the BTIC, so they are fetched in this cycle. Note that even though the first **bc** instruction may not have resolved by this point (we can assume it has), Gekko allows fetching from a second predicted branch stream. However, these instructions could not be dispatched until the previous branch has resolved.
3. In clock cycle 3, target instructions T2–T5 are fetched as T0 and T1 are dispatched.
4. In clock cycle 4, instruction 3, on which the second branch instruction depended, writes back and the branch prediction is proven incorrect. Even though T0 is in CQ1, from which it could be written back, it is not written back because the branch prediction was incorrect. All target instructions are flushed from their positions in the pipeline at the end of this clock cycle, as are any results in the rename registers.

After one clock cycle required to refetch the original instruction stream, instruction 5, the same instruction that was fetched in clock cycle 1, is brought back into the IQ from the instruction cache, along with three others (not all of which are shown).

6.4.2 Integer Unit Execution Timing

Gekko has two integer units. The IU1 can execute all integer instructions; and the IU2 can execute all integer instructions except multiply and divide instructions. As shown in Figure 6-2, each integer unit has one execute pipeline stage, thus when a multicycle integer instruction is being executed, no other integer instructions can begin to execute. Table 6-6 lists integer instruction latencies.

Most integer instructions have an execution latency of one clock cycle.

6.4.3 Floating-Point Unit Execution Timing

The floating-point unit on Gekko executes all floating-point instructions. Execution of most floating-point instructions is pipelined within the FPU, allowing up to three instructions to be executing in the FPU concurrently. While most floating-point instructions execute with three- or four-cycle latency, and one- or two-cycle throughput, two instructions (**fdivs** and **fdiv**) execute with latencies of 11 to 33 cycles. The **fdivs**, **fdiv**, **mtfsb0**, **mtfsb1**, **mtfsfi**, **mffs**, and **mtfsf** instructions block the floating-point unit pipeline until they complete execution, and thereby inhibit the dispatch of additional floating-point instructions. See Table 6-7 for floating-point instruction execution timing.

6.4.4 Effect of Floating-Point Exceptions on Performance

For the fastest and most predictable floating-point performance, all exceptions should be disabled in the FPSCR and MSR.

6.4.5 Load/Store Unit Execution Timing

The execution of most load and store instructions is pipelined. The LSU has two pipeline stages. The first is for effective address calculation and MMU translation and the second is for accessing data in the cache. Load and store instructions have a two-cycle latency and one-cycle throughput. For instructions that store FPR values (**stfd**, **stfs**, **psq_st**, and their variations), the data to be stored is prefetched from the source register during the first pipeline stage. In cases where this register is updated that same cycle, the instruction will stall to get the correct data, resulting in one additional cycle of latency.

If operands are misaligned, additional latency may be required either for an alignment exception to be taken or for additional bus accesses. Load instructions that miss in the cache block subsequent cache accesses during the cache line refill. Table 6-8 gives load and store instruction execution latencies.

6.4.6 Effect of Operand Placement on Performance

The PowerPC VEA states that the placement (location and alignment) of operands in memory may affect the relative performance of memory accesses, and in some cases affect it significantly. The effects memory operand placement has on performance are shown in Table 6-1.

The best performance is guaranteed if memory operands are aligned on natural boundaries. For the best performance across the widest range of implementations, the programmer should assume the performance model described in Chapter 3, “Operand Conventions” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

The effect of misalignment on memory access latency is the same for big- and little-endian addressing modes except for multiple and string operations that cause an alignment exception in little-endian mode.

Table 6-1. Performance Effects of Memory Operand Placement

Operand		Boundary Crossing			
Size	**Byte Alignment**	**None**	**8 Byte**	**Cache Block**	**Protection Boundary**
Integer					
4 byte	4	Optimal[1]	—	—	—
	< 4	Optimal	Good	Good	Good
2 byte	2	Optimal	—	—	—
	< 2	Optimal	Good	Good	Good
1 byte	1	Optimal	—	—	—
lmw, **stmw**[2]	4	Good [3]	Good	Good	Good
	< 4	Poor [4]	Poor	Poor	Poor
String [2]	—	Good	Good	Good	Good
Floating-Point					
8 byte	8	Optimal	—	—	—
	4	—	Good	Good	Good
	< 4	—	Poor	Poor	Poor
4 byte	4	Optimal	—	—	—
	< 4	Poor	Poor	Poor	Poor

Notes:

[1] Optimal means one EA calculation occurs.

[2] Not supported in little-endian mode, causes an alignment exception.

[3] Good means multiple EA calculations occur that may cause additional bus activities with multiple bus transfers.

[4] Poor means that an alignment exception occurs.

6.4.7 Integer Store Gathering

The Gekko performs store gathering for write-through operations to nonguarded space. It performs cache-inhibited stores to nonguarded space for 4-byte, word-aligned stores. These stores are combined in the LSU to form a double word and are sent out on the 60x bus as a single-beat operation. However, stores are gathered only if the successive stores meet the criteria and are queued and pending. Store gathering occurs regardless of the address order of the stores. Store gathering is enabled by setting HID0[SGE]. Stores can be gathered in both endian modes.

Store gathering is not done for the following:

- Cacheable store operations
- Stores to guarded cache-inhibited or write-through space
- Byte-reverse store operations
- **stwcx.** instructions
- ecowx instructions
- A store that occurs during a table search operation
- Floating-point store operations

If store gathering is enabled and the stores do not fall under the above categories, an **eieio** or **sync** instruction must be used to prevent two stores from being gathered.

6.4.8 System Register Unit Execution Timing

Most instructions executed by the SRU either directly access renamed registers or access or modify nonrenamed registers. They generally execute in a serial manner. Results from these instructions are not available to subsequent instructions until the instruction completes and is retired. See 6.3.2.7," for more information on serializing instructions executed by the SRU, and refer to Table 6-4 and Table 6-5 for SRU instruction execution timings.

6.5 Memory Performance Considerations

Because Gekko can have a maximum instruction throughput of three instructions per clock cycle, lack of memory bandwidth can affect performance. For the Gekko to maximize performance, it must be able to read and write data efficiently. If a system has multiple bus devices, one of them may experience long memory latencies while another bus master (for example, a direct-memory access controller) is using the external bus.

6.5.1 Caching and Memory Coherency

To minimize the effect of bus contention, the PowerPC architecture defines WIM bits that are used to configure memory regions as caching-enforced or caching-inhibited. Accesses to such memory locations never update the on-chip cache. If a cache-inhibited access hits the on-chip cache, the cache block is invalidated. If the cache block is marked modified, it is copied back to memory before being invalidated. Where caching is permitted, memory is configured as either write-back or write-through, which are described as follows:

- Write-back— Configuring a memory region as write-back lets a processor modify data in the cache without updating system memory. For such locations, memory updates occur only on modified cache block replacements, cache flushes, or when one processor needs data that is modified in another's cache. Therefore, configuring memory as write-back can help when bus traffic could cause bottlenecks, especially for multiprocessor systems and for regions in which data, such as local variables, is used often and is coupled closely to a processor.

If multiple devices use data in a memory region marked write-through, snooping must be enabled to allow the copy-back and cache invalidation operations necessary to ensure cache coherency. Gekko's snooping hardware keeps other devices from accessing invalid data. For example, when snooping is enabled, Gekko monitors transactions of other bus devices. For example, if another device needs data that is modified on Gekko's cache, the access is delayed so Gekko can copy the modified data to memory.

- Write-through—Store operations to memory marked write-through always update both system memory and the on-chip cache on cache hits. Because valid cache contents always match system memory marked write-through, cache hits from other devices do not cause modified data to be copied back as they do for locations marked write-back. However, all write operations are passed to the bus, which can limit performance. Load operations that miss the on-chip cache must wait for the external store operation.

  Write-through configuration is useful when cached data must agree with external memory (for example, video memory), when shared (global) data may be needed often, or when it is undesirable to allocate a cache block on a cache miss.

Chapter 3, "Gekko Instruction and Data Cache Operation" describes the caches, memory configuration, and snooping in detail.

6.5.2 Effect of TLB Miss

If a page address translation is not in a TLB, Gekko hardware searches the page tables and updates the TLB when a translation is found. Table 6-2 shows the estimated latency for the hardware TLB load for different cache configurations and conditions.

Table 6-2. TLB Miss Latencies

L1 Condition (Instruction and Data)	L2 Condition	Processor/System Bus Clock Ratio	Estimated Latency (Cycles)
100% cache hit	—	—	7
100% cache miss	100% cache hit	—	13
100% cache miss	100% cache miss	2.5:1 (6:3:3:3 memory)	62
100% cache miss	100% cache miss	4:1 (5:2:2:2 memory)	77

The PTE table search assumes a hit in the first entry of the primary PTEG.

6.6 Instruction Scheduling Guidelines

The performance of Gekko can be improved by avoiding resource conflicts and scheduling instructions to take fullest advantage of the parallel execution units. Instruction scheduling on Gekko can be improved by observing the following guidelines:

- To reduce mispredictions, separate the instruction that sets CR bits from the branch instruction that evaluates them. Because there can be no more than 12 instructions in the processor (with the instruction that sets CR in CQ0 and the dependent branch instruction in IQ5), there is no advantage to having more than 10 instructions between them.
- Likewise, when branching to a location specified by the CTR or LR, separate the **mtspr** instruction that initializes the CTR or LR from the dependent branch instruction. This ensures the register values are immediately available to the branch instruction.
- Schedule instructions such that two can be dispatched at a time.
- Schedule instructions to minimize stalls due to execution units being busy.
- Avoid scheduling high-latency instructions close together. Interspersing single-cycle latency instructions between longer-latency instructions minimizes the effect that instructions such as integer divide and multiply can have on throughput.
- Avoid using serializing instructions.
- Schedule instructions to avoid dispatch stalls:
  - — Six instructions can be tracked in the completion queue; therefore, only six instructions can be in the execute stages at any one time
  - — There are six GPR rename registers; therefore only six GPRs can be specified as destination operands at any time. If no rename registers are available, instructions cannot enter the execute stage and remain in the reservation station or instruction queue until they become available.

    NOTE: Load with update address instructions use two destination registers
  - — Similarly, there are six FPR rename registers, so only six FPR destination operands can be in the execute and complete stages at any time.

6.6.1 Branch, Dispatch, and Completion Unit Resource Requirements

This section describes the specific resources required to avoid stalls during branch resolution, instruction dispatching, and instruction completion.

6.6.1.1 Branch Resolution Resource Requirements

The following is a list of branch instructions and the resources required to avoid stalling the fetch unit in the course of branch resolution:

- The **bclr** instruction requires LR availability.
- The **bcctr** instruction requires CTR availability.
- Branch and link instructions require shadow LR availability.
- The “branch conditional on counter decrement and the CR” condition requires CTR availability or the CR condition must be false, and Gekko cannot execute instructions after an unresolved predicted branch when the BPU encounters a branch.
- A branch conditional on CR condition cannot be executed following an unresolved predicted branch instruction.

6.6.1.2 Dispatch Unit Resource Requirements

The following is a list of resources required to avoid stalls in the dispatch unit. IQ[0] and IQ[1] are the two dispatch entries in the instruction queue:

- Requirements for dispatching from IQ[0] are as follows:
  - Needed execution unit available
  - Needed GPR rename registers available
  - Needed FPR rename registers available
  - Completion queue is not full.
  - A completion-serialized instruction is not being executed.
- Requirements for dispatching from IQ[1] are as follows:
  - Instruction in IQ[0] must dispatch.
  - Instruction dispatched by IQ[0] is not completion- or refetch-serialized.
  - Needed execution unit is available (after dispatch from IQ[0]).
  - Needed GPR rename registers are available (after dispatch from IQ[0]).
  - Needed FPR rename register is available (after dispatch from IQ[0]).
  - Completion queue is not full (after dispatch from IQ[0]).

6.6.1.3 Completion Unit Resource Requirements

The following is a list of resources required to avoid stalls in the completion unit; note that the two completion entries are described as CQ[0] and CQ[1], where CQ[0] is the completion queue located at the end of the completion queue (see Figure 6-4).

- Requirements for completing an instruction from CQ[0] are as follows:
  - Instruction in CQ[0] must be finished.
  - Instruction in CQ[0] must not follow an unresolved predicted branch.
  - Instruction in CQ[0] must not cause an exception.
- Requirements for completing an instruction from CQ[1] are as follows:
  - Instruction in CQ[0] must complete in same cycle.
  - Instruction in CQ[1] must be finished.
  - Instruction in CQ[1] must not follow an unresolved predicted branch.
  - Instruction in CQ[1] must not cause an exception.
  - Instruction in CQ[1] must be an integer or load instruction.
  - Number of CR updates from both CQ[0] and CQ[1] must not exceed two.
  - Number of GPR updates from both CQ[0] and CQ[1] must not exceed two.
  - Number of FPR updates from both CQ[0] and CQ[1] must not exceed two.

6.7 Instruction Latency Summary

Table 6-3 through Table 6-8 on Page 6-34 list the latencies associated with instructions executed by each execution unit. Table 6-3 describes branch instruction latencies.

Table 6-3. Branch Instructions

Mnemonic	Primary	Extended	Latency
b[l][a]	18	—	Unless these instructions update either the CTR or the LR, branch operations are folded if they are either taken or predicted as taken. They fall through if they are not taken or predicted as not taken.
bc[l][a]	16	—	
bcctr[l]	19	528	
bclr[l]	19	16	

Table 6-4 lists system register instruction latencies.

Table 6-4. System Register Instructions

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
eieio	31	854	SRU	1	—
isync	19	150	SRU	2	Completion, refetch
mfmsr	31	83	SRU	1	—
mfspr (DBATs)	31	339	SRU	3	Execution
mfspr (IBATs)	31	339	SRU	3	—
mfspr (not I/DBATs)	31	339	SRU	1	Execution
mfsr	31	595	SRU	3	—
mfsrin	31	659	SRU	3	Execution
mftb	31	371	SRU	1	—
mtmsr	31	146	SRU	1	Execution
mtspr (DBATs)	31	467	SRU	2	Execution
mtspr (IBATs)	31	467	SRU	2	Execution
mtspr (not I/DBATs)	31	467	SRU	2	Execution
mtsr	31	210	SRU	2	Execution
mtsrin	31	242	SRU	2	Execution
mttb	31	467	SRU	1	Execution
rfi	19	50	SRU	2	Completion, refetch
sc	17	- -1	SRU	2	Completion, refetch
sync	31	598	SRU	3[1]	—
tlbsync [2]	31	566	—	—	

Notes:

[1] This assumes no pending stores in the store queue. If there are, the **sync** completes after they complete to memory. If broadcast is enabled on the 60x bus, **sync** completes only after a successful broadcast.

[2] **tlbsync** is dispatched only to the completion buffer (not to any execution unit) and is marked finished as it is dispatched. Upon retirement, it waits for an external $\overline{\text{TLBISYNC}}$ signal to be asserted. In most systems $\overline{\text{TLBISYNC}}$ is always asserted so the instruction is a no-op.

Table 6-5 lists condition register logical instruction latencies.

Table 6-5. Condition Register Logical Instructions

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
crand	19	257	SRU	1	Execution
crandc	19	129	SRU	1	Execution
creqv	19	289	SRU	1	Execution
crnand	19	225	SRU	1	Execution
crnor	19	33	SRU	1	Execution
cror	19	449	SRU	1	Execution
crorc	19	417	SRU	1	Execution
crxor	19	193	SRU	1	Execution
mcrf	19	0	SRU	1	Execution
mcrxr	31	512	SRU	1	Execution
mfcr	31	19	SRU	1	Execution
mtcrf	31	144	SRU	1	Execution

Table 6-6 shows integer instruction latencies. Note that the IU1 executes all integer arithmetic instructions—multiply, divide, shift, rotate, add, subtract, and compare. The IU2 executes all integer instructions except multiply and divide (that is, shift, rotate, add, subtract, and compare).

Table 6-6. Integer Instructions

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
addc[**o**][**.**]	31	10	IU1/IU2	1	—
adde[**o**][**.**]	31	138	IU1/IU2	1	Execution
addi	14	—	IU1/IU2	1	—
addic	12	—	IU1/IU2	1	—
addic.	13	—	IU1/IU2	1	—
addis	15	—	IU1/IU2	1	—
addme[**o**][**.**]	31	234	IU1/IU2	1	Execution
addze[**o**][**.**]	31	202	IU1/IU2	1	Execution

Table 6-6. Integer Instructions (Continued)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
add[**o**][**.**]	31	266	IU1/IU2	1	—
andc[**.**]	31	60	IU1/IU2	1	—
andi.	28	—	IU1/IU2	1	—
andis.	29	—	IU1/IU2	1	—
and[**.**]	31	28	IU1/IU2	1	—
cmp	31	0	IU1/IU2	1	—
cmpi	11	—	IU1/IU2	1	—
cmpl	31	32	IU1/IU2	1	—
cmpli	10	—	IU1/IU2	1	—
cntlzw[**.**]	31	26	IU1/IU2	1	—
divwu[**o**][**.**]	31	459	IU1	19	—
divw[**o**][**.**]	31	491	IU1	19	—
eqv[**.**]	31	284	IU1/IU2	1	—
extsb[**.**]	31	954	IU1/IU2	1	—
extsh[**.**]	31	922	IU1/IU2	1	—
mulhwu[**.**]	31	11	IU1/IU2	2,3,4,5,6	—
mulhw[**.**]	31	75	IU1/IU2	2,3,4,5	—
mulli	7	—	IU1	2,3	—
mull[**o**][**.**]	31	235	IU1	2,3,4,5	—
nand[**.**]	31	476	IU1/IU2	1	—
neg[**o**][**.**]	31	104	IU1/IU2	1	—
nor[**.**]	31	124	IU1/IU2	1	—
orc[**.**]	31	412	IU1/IU2	1	—
ori	24	—	IU1/IU2	1	—
oris	25	—	IU1/IU2	1	—
or[**.**]	31	444	IU1/IU2	1	—
rlwimi[**.**]	20	—	IU1/IU2	1	—
rlwinm[**.**]	21	—	IU1/IU2	1	—
rlwnm[**.**]	23	—	IU1/IU2	1	—
slw[**.**]	31	24	IU1/IU2	1	—
srawi[**.**]	31	824	IU1/IU2	1	—
sraw[**.**]	31	792	IU1/IU2	1	—

Table 6-6. Integer Instructions (Continued)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
srw[**.**]	31	536	IU1/IU2	1	—
subfc[**o**][**.**]	31	8	IU1/IU2	1	—
subfe[**o**][**.**]	31	136	IU1/IU2	1	Execution
subfic	8	—	IU1/IU2	1	—
subfme[**o**][**.**]	31	232	IU1/IU2	1	Execution
subfze[**o**][**.**]	31	200	IU1/IU2	1	Execution
subf[**.**]	31	40	IU1/IU2	1	—
tw	31	4	IU1/IU2	2	—
twi	3	—	IU1/IU2	2	—
xori	26	—	IU1/IU2	1	—
xoris	27	—	IU1/IU2	1	—
xor[**.**]	31	316	IU1/IU2	1	—

Table 6-7 shows latencies for floating-point instructions. Pipelined floating-point instructions are shown with number of clocks in each pipeline stage separated by dashes. Floating-point instructions with a single entry in the cycles column are not pipelined; when the FPU executes these nonpipelined instructions, it remains busy for the full duration of the instruction execution and is not available for subsequent instructions.

Table 6-7. Floating-Point Instructions

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
fabs[**.**]	63	264	FPU	1-1-1	—
fadds[**.**]	59	21	FPU	1-1-1	—
fadd[**.**]	63	21	FPU	1-1-1	—
fcmpo	63	32	FPU	1-1-1	—
fcmpu	63	0	FPU	1-1-1	—
fctiwz[**.**]	63	15	FPU	1-1-1	—
fctiw[**.**]	63	14	FPU	1-1-1	—
fdivs[**.**]	59	18	FPU	17	—
fdiv[**.**]	63	18	FPU	31	—
fmadds[**.**]	59	29	FPU	1-1-1	—
fmadd[**.**]	63	29	FPU	2-1-1	—
fmr[**.**]	63	72	FPU	1-1-1	—

Table 6-7. Floating-Point Instructions (Continued)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
fmsubs[.]	59	28	FPU	1-1-1	—
fmsub[.]	63	28	FPU	2-1-1	—
fmuls[.]	59	25	FPU	1-1-1	—
fmul[.]	63	25	FPU	2-1-1	—
fnabs[.]	63	136	FPU	1-1-1	—
fneg[.]	63	40	FPU	1-1-1	—
fnmadds[.]	59	31	FPU	1-1-1	—
fnmadd[.]	63	31	FPU	2-1-1	—
fnmsubs[.]	59	30	FPU	1-1-1	—
fnmsub[.]	63	30	FPU	2-1-1	—
fres[.]	59	24	FPU	2-1-1	—
frsp[.]	63	12	FPU	1-1-1	—
frsqrte[.]	63	26	FPU	2-1-1	—
fsel[.]	63	23	FPU	1-1-1	—
fsubs[.]	59	20	FPU	1-1-1	—
ps_abs[.]	4	264	FPU	1-1-1	—
ps_add[.]	4	21	FPU	1-1-1	—
ps_cmpo0	4	32	FPU	1-1-1	—
ps_cmpo1	4	96	FPU	1-1-1	—
ps_cmpu0	0	0	FPU	1-1-1	—
ps_cmpu1	4	64	FPU	1-1-1	—
ps_div[.]	4	18	FPU	17	—
ps_madd[.]	4	29	FPU	1-1-1	—
ps_madds0[.]	4	14	FPU	1-1-1	—
ps_madds1[.]	4	15	FPU	1-1-1	—
ps_merge00[.]	4	528	FPU	1-1-1	—
ps_merge01[.]	4	560	FPU	1-1-1	—
ps_merge10[.]	4	592	FPU	1-1-1	—
ps_merge_11[.]	4	624	FPU	1-1-1	—
ps_mr[.]	4	72	FPU	1-1-1	—
ps_msub[.]	4	28	FPU	1-1-1	—
ps_mul[.]	4	25	FPU	1-1-1	—

Table 6-7. Floating-Point Instructions (Continued)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
ps_muls0[.]	4	12	FPU	1-1-1	—
ps_muls1[.]	4	13	FPU	1-1-1	—
ps_nabs[.]	4	136	FPU	1-1-1	—
ps_neg[.]	4	40	FPU	1-1-1	—
ps_nmadd[.]	4	31	FPU	1-1-1	—
ps_nmsub[.]	4	30	FPU	1-1-1	—
ps_res[.]	4	24	FPU	2-1-1	—
ps_rsqrte[.]	4	26	FPU	2-1-1	—
ps_sel[.]	4	23	FPU	1-1-1	—
ps_sub[.]	4	20	FPU	1-1-1	—
ps_sum0[.]	4	10	FPU	1-1-1	—
ps_sum1[.]	4	11	FPU	1-1-1	—
fsub[.]	63	20	FPU	1-1-1	—
mcrfs	63	64	FPU	1-1-1	Execution
mffs[.]	63	583	FPU	1-1-1	Execution
mtfsb0[.]	63	70	FPU	3	—
mtfsb1[.]	63	38	FPU	3	—
mtfsfi[.]	63	134	FPU	3	—
mtfsf[.]	63	711	FPU	3	—

Table 6-8 shows load and store instruction latencies. Pipelined load/store instructions are shown with cycles of total latency and throughput cycles separated by a colon.

Table 6-8. Load and Store Instructions

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
dcbf	31	86	LSU	3:5[1]	Execution
dcbi	31	470	LSU	3:3[1]	Execution
dcbst	31	54	LSU	3:5[1]	Execution
dcbt	31	278	LSU	2:1	—
dcbtst	31	246	LSU	2:1	—
dcbz	31	1014	LSU	3:6[1, 2]	Execution
dcbz_l	4	1014	LSU	3:6[1]	Exceution

Table 6-8. Load and Store Instructions (Continued)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
eciwx	31	310	LSU	2:1	—
ecowx	31	438	LSU	2:1	—
icbi	31	982	LSU	3:4[1]	Execution
lbz	34	—	LSU	2:1	—
lbzu	35	—	LSU	2:1	—
lbzux	31	119	LSU	2:1	—
lbzx	31	87	LSU	2:1	—
lfd	50	—	LSU	2:1	—
lfdu	51	—	LSU	2:1	—
lfdux	31	631	LSU	2:1	—
lfdx	31	599	LSU	2:1	—
lfs	48	—	LSU	2:1	—
lfsu	49	—	LSU	2:1	—
lfsux	31	567	LSU	2:1	—
lfsx	31	535	LSU	2:1	—
lha	42	—	LSU	2:1	—
lhau	43	—	LSU	2:1	—
lhaux	31	375	LSU	2:1	—
lhax	31	343	LSU	2:1	—
lhbrx	31	790	LSU	2:1	—
lhz	40	—	LSU	2:1	—
lhzu	41	—	LSU	2:1	—
lhzux	31	311	LSU	2:1	—
lhzx	31	279	LSU	2:1	—
lmw	46	—	LSU	2 + n [3]	Completion, execution
lswi	31	597	LSU	2 + n [3]	Completion, execution
lswx	31	533	LSU	2 + n [3]	Completion, execution
lwarx	31	20	LSU	3:1	Execution
lwbrx	31	534	LSU	2:1	—
lwz	32	—	LSU	2:1	—
lwzu	33	—	LSU	2:1	—
lwzux	31	55	LSU	2:1	—

Table 6-8. Load and Store Instructions (Continued)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
lwzx	31	23	LSU	2:1	—
psq_l	56	—	LSU	3:1	—
psq_lu	57	—	LSU	3:1	—
psq_lux	4	38	LSU	3:1	—
psq_lx	4	6	LSU	3:1	—
psq_st	60	—	LSU	2:1	—
psq_stu	61	—	LSU	2:1	—
psq_stux	4	39	LSU	2:1	—
psq_stx	4	7	LSU	2:1	—
stb	38	—	LSU	2:1	—
stbu	39	—	LSU	2:1	—
stbux	31	247	LSU	2:1	—
stbx	31	215	LSU	2:1	—
stfd	54	—	LSU	2:1	—
stfdu	55	—	LSU	2:1	—
stfdux	31	759	LSU	2:1	—
stfdx	31	727	LSU	2:1	—
stfiwx	31	983	LSU	2:1	—
stfs	52	—	LSU	2:1	—
stfsu	53	—	LSU	2:1	—
stfsux	31	695	LSU	2:1	—
stfsx	31	663	LSU	2:1	—
sth	44	—	LSU	2:1	—
sthbrx	31	918	LSU	2:1	—
sthu	45	—	LSU	2:1	—
sthux	31	439	LSU	2:1	—
sthx	31	407	LSU	2:1	—
stmw	47	—	LSU	$2 + n^3$	Execution
stswi	31	725	LSU	$2 + n^3$	Execution
stswx	31	661	LSU	$2 + n^3$	Execution
stw	36	—	LSU	2:1	—

Table 6-8. Load and Store Instructions (Continued)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
stwbrx	31	662	LSU	2:1	—
stwcx.	31	150	LSU	8:8	Execution
stwu	37	—	LSU	2:1	—
stwux	31	183	LSU	2:1	—
stwx	31	151	LSU	2:1	—
tlbie	31	306	LSU	3:4[1]	Execution

Notes:

[1] For cache-ops, the first number indicates the latency in finishing a single instruction; the second indicates the throughput for back-to-back cache-ops. Throughput may be larger than the initial latency as more cycles may be needed to complete the instruction to the cache, which stays busy keeping subsequent cache-ops from executing.

[2] The throughput number of 6 cycles for **dcbz** assumes it is to nonglobal (M = 0) address space. For global address space, throughput is at least 11 cycles

[3] Load/store multiple/string instruction cycles are represented as a fixed number of cycles plus a variable number of cycles, where *n* is the number of words accessed by the instruction.

Chapter 7 Signal Descriptions

This chapter describes Gekko microprocessor's external signals. It contains a concise description of individual signals, showing behavior when the signal is asserted and negated and when the signal is an input and an output.

NOTE

A bar over a signal name indicates that the signal is active low—for example, $\overline{\text{ARTRY}}$ (address retry) and $\overline{\text{TS}}$ (transfer start). Active-low signals are referred to as asserted (active) when they are low and negated when they are high. Signals that are not active low, such as AP[0–3] (address bus parity signals) and TT[0–4] (transfer type signals) are referred to as asserted when they are high and negated when they are low.

Gekko's signals are grouped as follows:

- Address arbitration— Gekko uses these signals to arbitrate for address bus mastership.
- Address transfer start—These signals indicate that a bus master has begun a transaction on the address bus.
- Address transfer—These signals include the address bus and address parity signals. They are used to transfer the address and to ensure the integrity of the transfer.
- Transfer attribute—These signals provide information about the type of transfer, such as the transfer size and whether the transaction is bursted, write-through, or cache-inhibited.
- Address transfer termination—These signals are used to acknowledge the end of the address phase of the transaction. They also indicate whether a condition exists that requires the address phase to be repeated.
- Data arbitration— Gekko uses these signals to arbitrate for data bus mastership.
- Data transfer—These signals, which consist of the data bus and data parity, are used to transfer the data and to ensure the integrity of the transfer.
- Data transfer termination—Data termination signals are required after each data beat in a data transfer. In a single-beat transaction, the data termination signals also indicate the end of the tenure; while in burst accesses, the data termination signals apply to individual beats and indicate the end of the tenure only after the final data beat. They also indicate whether a condition exists that requires the data phase to be repeated.
- Interrupts/resets—These signals include the external interrupt signal, checkstop signals, and both soft reset and hard reset signals. They are used to interrupt and, under various conditions, to reset the processor.
- Processor status and control—These signals are used to set the reservation coherency bit, enable the time base, and other functions. They are also used in conjunction with such resources as secondary caches and the time base facility.
- Clock control—These signals determine the system clock frequency. They can also be used to synchronize multiprocessor systems.
- Test interface—The JTAG (IEEE 1149.1a-1993) interface and the common on-chip processor (COP) unit provide a serial interface to the system for performing board-level boundary-scan interconnect tests.

7.1 Signal Configuration

Figure 7-1 illustrates Gekko's signal configuration, showing how the signals are grouped. A pinout showing pin numbers is included in Gekko hardware specifications

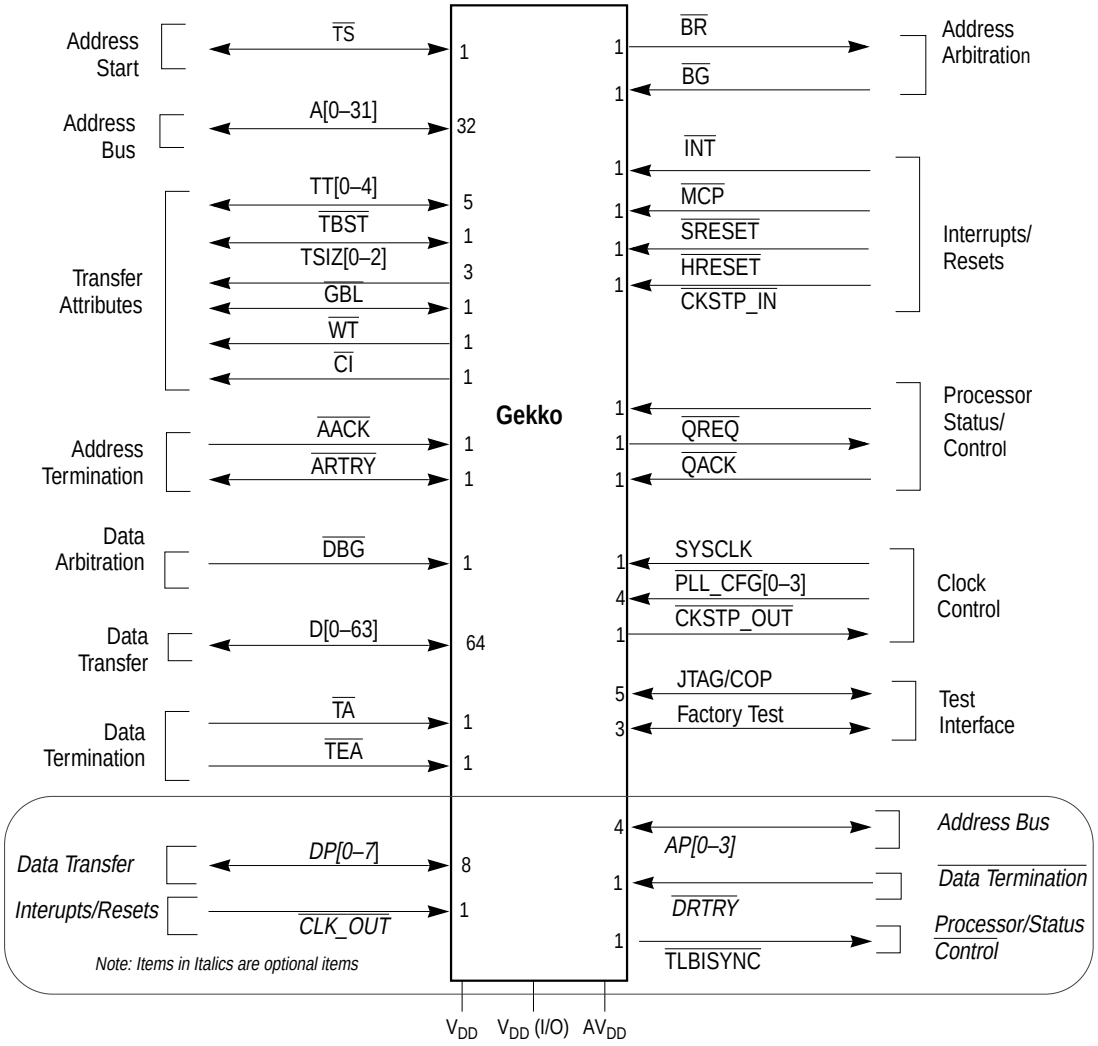


Figure 7-1. PowerPC Gekko Signal Groups

7.2 Signal Descriptions

This section describes individual signals on Gekko, grouped according to Figure 7-1.

NOTE: These sections summarize signal functions; Chapter 8, "Bus Interface Operation" describes many of these signals in greater detail, both with respect to how individual signals function and to how the groups of signals interact.

7.2.1 Address Bus Arbitration Signals

The address arbitration signals are the input and output signals Gekko uses to request the address bus, recognize when the request is granted, and indicate to other devices when mastership is granted.

For a detailed description of how these signals interact, see Section 8.3.1, “Address Bus Arbitration" on Page 8-9.

7.2.1.1 Bus Request ($\overline{BR}$)—Output

Following are the state meaning and timing comments for the $\overline{BR}$ output signal.

State Meaning

Asserted—Indicates that Gekko is requesting mastership of the address bus. Note that $\overline{BR}$ may be asserted for one or more cycles, and then de-asserted due to an internal cancellation of the bus request (for example, due to a load hit in the touch load buffer). See Section 8.3.1, “Address Bus Arbitration" on Page 8-9.

Negated—Indicates that Gekko is not requesting the address bus. Gekko may have no bus operation pending, it may be parked, or the $\overline{ARTRY}$ input was asserted on the previous bus clock cycle.

Timing Comments

Assertion—Occurs when Gekko is not parked and a bus transaction is needed. This may occur even if the two possible pipeline accesses have occurred. $\overline{BR}$ will also be asserted for one cycle during the execution of a **dcbz** instruction, and during the execution of a load instruction which hits in the touch load buffer.

Negation—Occurs for at least one bus clock cycle after an accepted, qualified bus grant (see $\overline{BG}$), even if another transaction is pending. It is also negated for at least one bus clock cycle when the assertion of $\overline{ARTRY}$ is detected on the bus.

7.2.1.2 Bus Grant ($\overline{BG}$)—Input

Following are the state meaning and timing comments for the $\overline{BG}$ input signal.

State Meaning

Asserted—Indicates that Gekko may, with proper qualification, assume mastership of the address bus. A qualified bus grant occurs when $\overline{BG}$ is asserted and $\overline{ARTRY}$ is not asserted the bus cycle following the assertion of $\overline{AACK}$. The $\overline{ARTRY}$ signal is driven by Gekko or other bus masters. If Gekko is parked, $\overline{BR}$ need not be asserted for the qualified bus grant. See Section 8.3.1, “Address Bus Arbitration" on Page 8-9.

Negated— Indicates that Gekko is not the next potential address bus master.

Timing Comments

Assertion—May occur at any time to indicate Gekko can use the address bus. After Gekko assumes bus mastership, it does not check for a qualified bus grant again until the cycle during which the address bus tenure completes (assuming it has another transaction to run). Gekko does not accept a $\overline{BG}$ in the cycles between the assertion of any $\overline{TS}$ and $\overline{AACK}$.

Negation—May occur at any time to indicate Gekko cannot use the bus. Gekko may still assume bus mastership on the bus clock cycle of the negation of $\overline{BG}$ because during the previous cycle $\overline{BG}$ indicated to Gekko that it could take mastership (if qualified).

7.2.2 Address Transfer Start Signals

Address transfer start signals are input and output signals that indicate that an address bus transfer has begun. The transfer start ($\overline{TS}$) signal identifies the operation as a memory transaction.

For detailed information about how $\overline{TS}$ interacts with other signals, refer to Section 8.3.2, “Address Transfer" on Page 8-11.

7.2.2.1 Transfer Start ($\overline{TS}$)

The $\overline{TS}$ signal is both an input and an output signal on Gekko.

7.2.2.1.1 Transfer Start ($\overline{TS}$)—Output

Following are the state meaning and timing comments for the $\overline{TS}$ output signal.

State Meaning Asserted—Indicates that Gekko has begun a memory bus transaction and that the address bus and transfer attribute signals are valid. When asserted with the appropriate TT[0–4] signals it is also an implied data bus request for a memory transaction (unless it is an address-only operation).

Negated—Indicates that no bus transaction is occurring during normal operation.

Timing Comments Assertion—May occur in a bus cycle following a qualified bus grant.
Negation—Occurs one bus clock cycle after $\overline{TS}$ is asserted.
High Impedance—Occurs the bus cycle following $\overline{AACK}$.

7.2.2.1.2 Transfer Start ($\overline{TS}$)—Input

Following are the state meaning and timing comments for the $\overline{TS}$ input signal.

State Meaning Asserted—Indicates that another master has begun a bus transaction and that the address bus and transfer attribute signals are valid for snooping (see $\overline{GBL}$).

Negated—Indicates that no bus transaction is occurring.

Timing Comments Assertion—May occur in a bus cycle following a qualified bus grant.
Negation—Must occur one bus clock cycle after $\overline{TS}$ is asserted.

7.2.3 Address Transfer Signals

The address transfer signals are used to transmit the address and to generate and monitor parity for the address transfer. For a detailed description of how these signals interact, refer to Section 8.3.2, “Address Transfer" on Page 8-11.

7.2.3.1 Address Bus (A[0–31])

The address bus (A[0–31]) consists of 32 signals that are both input and output signals.

7.2.3.1.1 Address Bus (A[0–31])—Output

Following are the state meaning and timing comments for the A[0–31] output signals.

State Meaning Asserted/Negated—Represents the physical address (real address in the architecture specification) of the data to be transferred. On burst transfers, the address bus presents the double-word-aligned address containing the critical code/data that missed the cache on a read operation, or the first double word of the cache line on a write operation. Note that the address output during burst operations is not incremented. See SSection 8.3.2, “Address Transfer" on Page 8-11.

Timing Comments Assertion/Negation—Occurs on the bus clock cycle after a qualified bus grant (coincides with assertion of $\overline{TS}$).

High Impedance—Occurs one bus clock cycle after $\overline{AACK}$ is asserted.

7.2.3.1.2 Address Bus (A[0–31])—Input

Following are the state meaning and timing comments for the A[0–31] input signals.

State Meaning Asserted/Negated—Represents the physical address of a snoop operation.

Timing Comments Assertion/Negation—Must occur on the same bus clock cycle as the assertion of $\overline{TS}$; is sampled by Gekko only on this cycle.

7.2.3.2 Address Bus Parity (AP[0–3]) (N/A on Gekko)

The address bus parity (AP[0–3]) signals are both input and output signals reflecting one bit of odd-byte parity for each of the 4 bytes of address when a valid address is on the bus. Address Bus Parity (AP[0–3])—Output

Following are the state meaning and timing comments for the AP[0–3] output signals.

State Meaning Asserted/Negated—Represents odd parity for each of the 4 bytes of the physical address for a transaction. Odd parity means that an odd number of bits, including the parity bit, are driven high. The signal assignments correspond to the following:

AP0 A[0–7]
AP1 A[8–15]
AP2 A[16–23]
AP3 A[24–31]

Timing Comments Assertion/Negation—The same as A[0–31].
High Impedance—The same as A[0–31].

7.2.3.2.1 Address Bus Parity (AP[0–3])—Input

Following are the state meaning and timing comments for the AP[0–3] input signal.

State Meaning Asserted/Negated—Represents odd parity for each of the 4 bytes of the physical address for snooping operations. Detected even parity causes the processor to take a machine check exception or enter the checkstop state if address parity checking is enabled in the HID0 register; see Section 2.1.2.2, “Hardware Implementation-Dependent Register 0" on

Page 2-8.

Timing Comments Assertion/Negation—The same as A[0–31].

7.2.4 Address Transfer Attribute Signals

The transfer attribute signals are a set of signals that further characterize the transfer—such as the size of the transfer, whether it is a read or write operation, and whether it is a burst or single-beat transfer. For a detailed description of how these signals interact, see Section 8.3.2, “Address Transfer" on Page 8-11.

NOTE: Some signal functions vary depending on whether the transaction is a memory access or an I/O access.

7.2.4.1 Transfer Type (TT[0–4])

The transfer type (TT[0–4]) signals consist of five input/output signals on Gekko. For a complete description of TT[0–4] signals and for transfer type encodings, see Table 7-1.

7.2.4.1.1 Transfer Type (TT[0–4])—Output

Following are the state meaning and timing comments for the TT[0–4] output signals on Gekko.

State Meaning Asserted/Negated—Indicates the type of transfer in progress.

Timing Comments Assertion/Negation/High Impedance—The same as A[0–31].

7.2.4.1.2 Transfer Type (TT[0–4])—Input

Following are the state meaning and timing comments for the TT[0–4] input signals on Gekko.

State Meaning Asserted/Negated—Indicates the type of transfer in progress (see Table 7-2 on Page 7-8).

Timing Comments Assertion/Negation—The same as A[0–31].

Table 7-1 describes the transfer encodings for an Gekko bus master.

Table 7-1. Transfer Type Encodings for PowerPC Gekko Bus Master

Gekko Bus Master Transaction	Transaction Source	TT0	TT1	TT2	TT3	TT4	60x Bus Specification Command	Transaction
Address only[1]	**dcbst**	0	0	0	0	0	Clean block	Address only
Address only[1]	**dcbf**	0	0	1	0	0	Flush block	Address only
Address only[1]	**sync**	0	1	0	0	0	**sync**	Address only
Address only[1]	**dcbz** or **dcbi**	0	1	1	0	0	Kill block	Address only
Address only[1]	**eieio**	1	0	0	0	0	**eieio**	Address only
Single-beat write (non$\overline{\text{GBL}}$)	**ecowx**	1	0	1	0	0	External control word write	Single-beat write
N/A	N/A	1	1	0	0	0	TLB invalidate	Address only
Single-beat read (non$\overline{\text{GBL}}$)	**eciwx**	1	1	1	0	0	External control word read	Single-beat read
N/A	N/A	0	0	0	0	1	**lwarx** reservation set	Address only
N/A	N/A	0	0	1	0	1	Reserved	—
N/A	N/A	0	1	0	0	1	**tlbsync**	Address only
N/A	N/A	0	1	1	0	1	**icbi**	Address only
N/A	N/A	1	X	X	0	1	Reserved	—

Table 7-1. Transfer Type Encodings for PowerPC Gekko Bus Master (Continued)

Gekko Bus Master Transaction	Transaction Source	TT0	TT1	TT2	TT3	TT4	60x Bus Specification Command	Transaction
Single-beat write	Caching-inhibited or write-through store, DMA, or write gather pipe	0	0	0	1	0	Write-with-flush	Single-beat write or burst
Burst (non$\overline{GBL}$)	Cast-out, or snoop copyback	0	0	1	1	0	Write-with-kill	Burst
Single-beat read	Caching-inhibited load or instruction fetch, or DMA	0	1	0	1	0	Read	Single-beat read or burst
Burst	Load miss, store miss, or instruction fetch	0	1	1	1	0	Read-with-intent-to-modify	Burst
Single-beat write	**stwcx.**	1	0	0	1	0	Write-with-flush-atomic	Single-beat write
N/A	N/A	1	0	1	1	0	Reserved	N/A
Single-beat read	**lwarx** (caching-inhibited load)	1	1	0	1	0	Read-atomic	Single-beat read or burst
Burst	**lwarx** (load miss)	1	1	1	1	0	Read-with-intent-to-modify-atomic	Burst
N/A	N/A	0	0	0	1	1	Reserved	—
N/A	N/A	0	0	1	1	1	Reserved	—
N/A	DMA	0	1	0	1	1	Read-with-no-intent-to-cache	Single-beat read or burst
N/A	N/A	0	1	1	1	1	Reserved	—
N/A	N/A	1	X	X	1	1	Reserved	—

Note: [1]Address-only transaction occurs if enabled by setting HID0[ABE] bit to 1.

Table 7-2 describes the 60x bus specification transfer encodings and Gekko bus snoop response on an address hit.

Table 7-2. PowerPC Gekko Snoop Hit Response

60x Bus Specification Command	Transaction	TT0	TT1	TT2	TT3	TT4	PowerPC Gekko Bus Snooper; Action on Hit
Clean block	Address only	0	0	0	0	0	N/A
Flush block	Address only	0	0	1	0	0	N/A
sync	Address only	0	1	0	0	0	N/A
Kill block	Address only	0	1	1	0	0	Flush, cancel reservation
eieio	Address only	1	0	0	0	0	N/A
External control word write	Single-beat write	1	0	1	0	0	N/A
TLB Invalidate	Address only	1	1	0	0	0	N/A
External control word read	Single-beat read	1	1	1	0	0	N/A
lwarx reservation set	Address only	0	0	0	0	1	N/A
Reserved	—	0	0	1	0	1	N/A
tlbsync	Address only	0	1	0	0	1	N/A
icbi	Address only	0	1	1	0	1	N/A
Reserved	—	1	X	X	0	1	N/A
Write-with-flush	Single-beat write or burst	0	0	0	1	0	Flush, cancel reservation
Write-with-kill	Single-beat write or burst	0	0	1	1	0	Kill, cancel reservation
Read	Single-beat read or burst	0	1	0	1	0	Clean or flush
Read-with-intent-to-modify	Burst	0	1	1	1	0	Flush
Write-with-flush-atomic	Single-beat write	1	0	0	1	0	Flush, cancel reservation
Reserved	N/A	1	0	1	1	0	N/A
Read-atomic	Single-beat read or burst	1	1	0	1	0	Clean or flush
Read-with-intent-to modify-atomic	Burst	1	1	1	1	0	Flush
Reserved	—	0	0	0	1	1	N/A
Reserved	—	0	0	1	1	1	N/A
Read-with-no-intent-to-cache	Single-beat read or burst	0	1	0	1	1	Clean
Reserved	—	0	1	1	1	1	N/A
Reserved	—	1	X	X	1	1	N/A

7.2.4.2 Transfer Size (TSIZ[0–2])—Output

Following are the state meaning and timing comments for the transfer size (TSIZ[0–2]) output signals on Gekko.

State Meaning Asserted/Negated—For memory accesses, these signals along with $\overline{\text{TBST}}$, indicate the data transfer size for the current bus operation, as shown in Table 7-3.
Table 8-6 shows how the transfer size signals are used with the address signals for aligned transfers.
Table 8-4 shows how the transfer size signals are used with the address signals for misaligned transfers.

NOTE: Gekko does not generate all possible TSIZ[0–2] encodings.

For external control instructions (**eciwx** and **ecowx**), TSIZ[0–2] are used to output bits 29–31 of the external access register (EAR), which are used to form the resource ID ($\overline{\text{TBST}}$||TSIZ0–TSIZ2).

Timing Comments Assertion/Negation—The same as A[0–31].
High Impedance—The same as A[0–31].

Table 7-3. Data Transfer Size

$\overline{\text{TBST}}$	TSIZ[0–2]	Transfer Size
Asserted	010	Burst (32 bytes)
Negated	000	8 bytes
Negated	001	1 byte
Negated	010	2 bytes
Negated	011	3 bytes
Negated	100	4 bytes
Negated	101	5 bytes[1]
Negated	110	6 bytes[1]
Negated	111	7 bytes[1]
Note: [1]Not generated by Gekko.		

7.2.4.3 Transfer Burst ($\overline{\text{TBST}}$)

The transfer burst ($\overline{\text{TBST}}$) signal is an input/output signal on Gekko.

7.2.4.3.1 Transfer Burst ($\overline{\text{TBST}}$)—Output

Following are the state meaning and timing comments for the $\overline{\text{TBST}}$ output signal.

State Meaning Asserted—Indicates that a burst transfer is in progress.

Negated—Indicates that a burst transfer is not in progress.

For external control instructions (**eciwx** and **ecowx**), $\overline{\text{TBST}}$ is used to output bit 28 of the EAR, which is used to form the resource ID ($\overline{\text{TBST}}$||TSIZ0–TSIZ2).

Timing Comments Assertion/Negation—The same as A[0–31].
High Impedance—The same as A[0–31].

7.2.4.3.2 Transfer Burst ($\overline{\text{TBST}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{TBST}}$ input signal.

State Meaning Asserted/Negated—Used when snooping for single-beat reads (read with no intent to cache).

Timing Comments Assertion/Negation—The same as A[0–31].

7.2.4.4 Cache Inhibit ($\overline{CI}$)—Output

The cache inhibit ($\overline{CI}$) signal is an output signal on Gekko. Following are the state meaning and timing comments for the $\overline{CI}$ signal.

State Meaning Asserted—Indicates that a single-beat transfer will not be cached, reflecting the setting of the I bit for the block or page that contains the address of the current transaction.

Negated—Indicates that a burst transfer will allocate an Gekko data cache block.

Timing Comments Assertion/Negation—The same as A[0–31].
High Impedance—The same as A[0–31].

7.2.4.5 Write-Through ($\overline{WT}$)—Output

The write-through ($\overline{WT}$) signal is an output signal on Gekko. Following are the state meaning and timing comments for the $\overline{WT}$ signal.

State Meaning Asserted—Indicates that a single-beat write transaction is write-through, reflecting the value of the W bit for the block or page that contains the address of the current transaction. Assertion during a read operation indicates instruction fetching.

Negated—Indicates that a write transaction is not write-through; during a read operation negation indicates a data load.

Timing Comments Assertion/Negation—The same as A[0–31].
High Impedance—The same as A[0–31].

7.2.4.6 Global ($\overline{GBL}$)

The global ($\overline{GBL}$) signal is an input/output signal on Gekko.

7.2.4.6.1 Global ($\overline{GBL}$)—Output

Following are the state meaning and timing comments for the $\overline{GBL}$ output signal.

State Meaning Asserted—Indicates that a transaction is global, reflecting the setting of the M bit for the block or page that contains the address of the current transaction (except in the case of copy-back operations and instruction fetches, which are nonglobal.)

Negated—Indicates that a transaction is not global.

Timing Comments Assertion/Negation—The same as A[0–31].
High Impedance—The same as A[0–31].

7.2.4.6.2 Global ($\overline{GBL}$)—Input

Following are the state meaning and timing comments for the $\overline{GBL}$ input signal.

State Meaning Asserted—Indicates that a transaction must be snooped by Gekko.

Negated—Indicates that a transaction is not snooped by Gekko.

Timing Comments Assertion/Negation—The same as A[0–31].

7.2.5 Address Transfer Termination Signals

The address transfer termination signals are used to indicate either that the address phase of the transaction has completed successfully or must be repeated, and when it should be terminated. For detailed information about how these signals interact, see Chapter 8.

7.2.5.1 Address Acknowledge ($\overline{\text{AACK}}$)—Input

The address acknowledge ($\overline{\text{AACK}}$) signal is an input-only signal on Gekko. Following are the state meaning and timing comments for the $\overline{\text{AACK}}$ signal.

State Meaning Asserted—Indicates that the address phase of a transaction is complete. The address bus will go to a high-impedance state on the next bus clock cycle. Gekko samples $\overline{\text{ARTRY}}$ on the bus clock cycle following the assertion of $\overline{\text{AACK}}$.

Negated—(During address bus tenure) indicates that the address bus and the transfer attributes must remain driven.

Timing Comments Assertion—May occur as early as the bus clock cycle after $\overline{\text{TS}}$ is asserted; assertion can be delayed to allow adequate address access time for slow devices. For example, if an implementation supports slow snooping devices, an external arbiter can postpone the assertion of $\overline{\text{AACK}}$.

Negation—Must occur one bus clock cycle after the assertion of $\overline{\text{AACK}}$.

7.2.5.2 Address Retry ($\overline{\text{ARTRY}}$)

The address retry ($\overline{\text{ARTRY}}$) signal is both an input and output signal on Gekko.

7.2.5.2.1 Address Retry ($\overline{\text{ARTRY}}$)—Output

Following are the state meaning and timing comments for the $\overline{\text{ARTRY}}$ output signal.

State Meaning Asserted—Indicates that Gekko detects a condition in which a snooped address tenure must be retried. If Gekko needs to update memory as a result of the snoop that caused the retry, Gekko asserts $\overline{\text{BR}}$ the second cycle after $\overline{\text{AACK}}$ if $\overline{\text{ARTRY}}$ is asserted.

High Impedance—Indicates that Gekko does not need the snooped address tenure to be retried.

Timing Comments Assertion—Asserted the third bus cycle following the assertion of $\overline{\text{TS}}$ if a retry is required.

Negation—Occurs the second bus cycle after the assertion of $\overline{\text{AACK}}$. Since this signal may be simultaneously driven by multiple devices, it negates in a unique fashion. First the buffer goes to high impedance for a minimum of one-half processor cycle (dependent on the clock mode), then it is driven negated for one-half bus cycle before returning to high impedance.

This special method of negation may be disabled by setting precharge disable in HID0.

7.2.5.2.2 Address Retry ($\overline{\text{ARTRY}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{ARTRY}}$ input signal.

State Meaning Asserted—If Gekko is the address bus master, $\overline{\text{ARTRY}}$ indicates that Gekko must retry the preceding address tenure and immediately negate $\overline{\text{BR}}$ (if asserted). If the associated data tenure has already started, Gekko also aborts the data tenure immediately, even if the burst data has been

received. If Gekko is not the address bus master, this input indicates that Gekko should immediately negate $\overline{BR}$ to allow an opportunity for a copy-back operation to main memory after a snooping bus master asserts $\overline{ARTRY}$. Note that the subsequent address presented on the address bus may not be the same one associated with the assertion of the $\overline{ARTRY}$ signal.

Negated/High Impedance—Indicates that Gekko does not need to retry the last address tenure.

Timing Comments Assertion—May occur as early as the second cycle following the assertion of $\overline{TS}$, and must occur by the bus clock cycle immediately following the assertion of $\overline{AACK}$ if an address retry is required.

Negation—Must occur two bus clock cycles after the assertion of $\overline{AACK}$.

7.2.6 Data Bus Arbitration Signals

Like the address bus arbitration signals, data bus arbitration signals maintain an orderly process for determining data bus mastership. Note that there is no data bus arbitration signal equivalent to the address bus arbitration signal $\overline{BR}$ (bus request), because, except for address-only transactions, $\overline{TS}$ implies data bus requests. For a detailed description on how these signals interact, see Section 8.4.1, “Data Bus Arbitration" on Page 8-18.

7.2.6.1 Data Bus Grant ($\overline{DBG}$)—Input

The data bus grant ($\overline{DBG}$) signal is an input-only signal on Gekko. Following are the state meaning and timing comments for the $\overline{DBG}$ signal.

State Meaning Asserted—Indicates that Gekko may, with the proper qualification, assume mastership of the data bus. Gekko derives a qualified data bus grant when $\overline{DBG}$ is asserted and $\overline{ARTRY}$ is negated; that is, there is no outstanding attempt to perform an $\overline{ARTRY}$ of the associated address tenure.

Negated—Indicates that Gekko must hold off its data tenures.

Timing Comments Assertion—May occur any time to indicate Gekko is free to take data bus mastership. It is not sampled until $\overline{TS}$ is asserted.

Negation—May occur at any time to indicate Gekko cannot assume data bus mastership.

7.2.7 Data Transfer Signals

Like the address transfer signals, the data transfer signals are used to transmit data and to generate and monitor parity for the data transfer. For a detailed description of how the data transfer signals interact, see Chapter 8.

7.2.7.1 Data Bus (DH[0–31], DL[0–31])

The data bus (DH[0–3]1 and DL[0–31]) consists of 64 signals that are both inputs and outputs on Gekko. Following are the state meaning and timing comments for the DH and DL signals.

State Meaning The data bus has two halves—data bus high (DH) and data bus low (DL). See Table 7-4 for the data bus lane assignments.

Timing Comments The data bus is driven once for noncached transactions and four times for cache transactions (bursts).

Table 7-4. Data Bus Lane Assignments

Data Bus Signals	Byte Lane
DH[0–7]	0
DH[8–15]	1
DH[16–23]	2
DH[24–31]	3
DL[0–7]	4
DL[8–15]	5
DL[16–23]	6
DL[24–31]	7

7.2.7.1.1 Data Bus (DH[0–31], DL[0–31])—Output

Following are the state meaning and timing comments for the DH and DL output signals.

State Meaning Asserted/Negated—Represents the state of data during a data write. Byte lanes not selected for data transfer will not supply valid data.

Timing Comments Assertion/Negation—Initial beat coincides with the bus cycle following a qualified $\overline{\text{DBG}}$ and, for bursts, transitions on the bus clock cycle following each assertion of $\overline{\text{TA}}$.

High Impedance—Occurs on the bus clock cycle after the final assertion of $\overline{\text{TA}}$, following the assertion of $\overline{\text{TEA}}$, or in certain $\overline{\text{ARTRY}}$ cases.

7.2.7.1.2 Data Bus (DH[0–31], DL[0–31])—Input

Following are the state meaning and timing comments for the DH and DL input signals.

State Meaning Asserted/Negated—Represents the state of data during a data read transaction.

Timing Comments Assertion/Negation—Data must be valid on the same bus clock cycle that $\overline{\text{TA}}$ is asserted.

7.2.7.2 Data Bus Parity (DP[0–8]) (N/A on Gekko)

The eight data bus parity (DP[0–7]) signals are both output and input signals.

7.2.7.2.1 Data Bus Parity (DP[0–7])—Output

Following are the state meaning and timing comments for the DP output signals.

State Meaning Asserted/Negated—Represents odd parity for each of the 8 bytes of data write transactions. Odd parity means that an odd number of bits, including the parity bit, are driven high. The generation of parity is enabled through HID0. The signal assignments are listed in Table 7-5.

Timing Comments Assertion/Negation—The same as DL[0–31].
High Impedance—The same as DL[0–31].

Table 7-5. DP[0–7] Signal Assignments

Signal Name	Signal Assignments
DP0	DH[0–7]
DP1	DH[8–15]
DP2	DH[16–23]
DP3	DH[24–31]
DP4	DL[0–7]
DP5	DL[8–15]
DP6	DL[16–23]
DP7	DL[24–31]

7.2.7.2.2 Data Bus Parity (DP[0–7])—Input

Following are the state meaning and timing comments for the DP input signals.

State Meaning Asserted/Negated—Represents odd parity for each byte of read data. Parity is checked on all data byte lanes, regardless of the size of the transfer. Detected even parity causes a checkstop if data parity errors are enabled in the HID0 register.

Timing Comments Assertion/Negation—The same as DL[0–31].

7.2.8 Data Transfer Termination Signals

Data termination signals are required after each data beat in a data transfer. Note that in a single-beat transaction, the data termination signals also indicate the end of the tenure, while in burst accesses, the data termination signals apply to individual beats and indicate the end of the tenure only after the final data beat.

For a detailed description of how these signals interact, see Chapter 8.

7.2.8.1 Transfer Acknowledge ($\overline{\text{TA}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{TA}}$ signal.

State Meaning Asserted— Indicates that a single-beat data transfer completed successfully or that a data beat in a burst transfer completed successfully. Note that $\overline{\text{TA}}$ must be asserted for each data beat in a burst transaction. For more information, see Chapter 8.

Negated—If Gekko is the data bus master Gekko must continue to drive the

data for the current write or must wait to sample the data for reads until $\overline{TA}$ is asserted.

Timing Comments Assertion—Must not occur before $\overline{AACK}$ for the current transaction (if the address retry mechanism is to be used to prevent invalid data from being used by the processor); otherwise, assertion may occur at any time Gekko while Gekko is the data bus master. The system can withhold assertion of $\overline{TA}$ to indicate that Gekko should insert wait states to extend the duration of the data beat.

Negation—Must occur after the bus clock cycle of the final (or only) data beat of the transfer. For a burst transfer, the system can assert $\overline{TA}$ for one bus clock cycle and then negate it to advance the burst transfer to the next beat and insert wait states during the next beat.

7.2.8.2 Data Retry ($\overline{DRTRY}$)—Input (N/A on Gekko)

Following are the state meaning and timing comments for the $\overline{DRTRY}$ signal.

State Meaning Asserted—Indicates that Gekko must invalidate the data from the previous read operation.

Negated—Indicates that data presented with $\overline{TA}$ on the previous read operation is valid. Note that $\overline{DRTRY}$ is ignored for write transactions.

Timing Comments Assertion—Must occur during the bus clock cycle immediately after $\overline{TA}$ is asserted if a retry is required. The $\overline{DRTRY}$ signal may be held asserted for multiple bus clock cycles. When $\overline{DRTRY}$ is negated, data must have been valid on the previous clock with $\overline{TA}$ asserted.

Negation—Must occur during the bus clock cycle after a valid data beat. This may occur several cycles effectively extending the data bus tenure.

Start-up—The $\overline{DRTRY}$ signal is sampled at the negation of $\overline{HRESET}$; if $\overline{DRTRY}$ is asserted, no-$\overline{DRTRY}$ mode is selected. If $\overline{DRTRY}$ is negated at start-up, $\overline{DRTRY}$ is enabled.

7.2.8.3 Transfer Error Acknowledge ($\overline{TEA}$)—Input

Following are the state meaning and timing comments for the $\overline{TEA}$ signal.

State Meaning Asserted—Indicates that a bus error occurred. Causes a machine check exception (and possibly causes the processor to enter checkstop state if machine check enable bit is cleared (MSR[ME] = 0)). For more information, see Section 4.5.2.2, “Checkstop State (MSR[ME] = 0)" on Page 4-17. Assertion terminates the current transaction; that is, assertion of $\overline{TA}$ is ignored. The assertion of $\overline{TEA}$ causes data bus tenure to be dropped. However, data entering the GPR or the cache are not invalidated. (Note that the term ‘exception’ is also referred to as ‘interrupt’ in the architecture specification.)

Negated—Indicates that no bus error was detected.

Timing Comments Assertion—May be asserted while Gekko is the data bus master, and the cycle after $\overline{TA}$ during a read operation. $\overline{TEA}$ should be asserted for one cycle only.

Negation—$\overline{TEA}$ must be negated no later than the end of the data bus tenure.

7.2.9 System Status Signals

Most system status signals are input signals that indicate when exceptions are received, when checkstop conditions have occurred, and when Gekko must be reset.

7.2.9.1 Interrupt ($\overline{\text{INT}}$)— Input

Following are the state meaning and timing comments for the $\overline{\text{INT}}$ signal.

State Meaning Asserted—Gekko initiates an interrupt if MSR[EE] is set; otherwise, Gekko ignores the interrupt. To guarantee that Gekko will take the external interrupt, $\overline{\text{INT}}$ must be held active until Gekko takes the interrupt; otherwise, whether Gekko takes an external interrupt depends on whether the MSR[EE] bit was set while the $\overline{\text{INT}}$ signal was held active.

Negated—Indicates that normal operation should proceed. See Chapter 8.

Timing Comments Assertion—May occur at any time and may be asserted asynchronously to the input clocks. The $\overline{\text{INT}}$ input is level-sensitive.
Negation—Should not occur until interrupt is taken.

7.2.9.2 Machine Check Interrupt ($\overline{\text{MCP}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{MCP}}$ signal.

State Meaning Asserted—Gekko initiates a machine check interrupt operation if MSR[ME] and HID0[EMCP] are set; if MSR[ME] is cleared and HID0[EMCP] is set, Gekko must terminate operation by internally gating off all clocks, and releasing all outputs to the high-impedance state. If HID0[EMCP] is cleared, Gekko ignores the interrupt condition. The $\overline{\text{MCP}}$ signal must be held asserted for two bus clock cycles.

Negated—Indicates that normal operation should proceed. See Section 8.8.1, “External Interrupts" on Page 8-36.

Timing Comments Assertion—May occur at any time and may be asserted asynchronously to the input clocks. The $\overline{\text{MCP}}$ input is negative edge-sensitive.

Negation—May be negated two bus cycles after assertion.

7.2.9.3 Checkstop Input ($\overline{\text{CKSTP\_IN}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{CKSTP\_IN}}$ signal.

State Meaning Asserted—Indicates that Gekko must terminate operation by internally gating off all clocks, and release all outputs to the high-impedance state. Once $\overline{\text{CKSTP\_IN}}$ has been asserted it must remain asserted until the system has been reset.

Negated—Indicates that normal operation should proceed. See Chapter 8.

Timing Comments Assertion—May occur at any time and may be asserted asynchronously to the input clocks.

Negation—May occur any time after the system reset.

7.2.9.4 Checkstop Output ($\overline{\text{CKSTP\_OUT}}$)—Output

Note that the $\overline{\text{CKSTP\_OUT}}$ signal is an open-drain type output, and requires an external pull-up resistor (for example, 10 k to V_{dd}) to assure proper de-assertion of the $\overline{\text{CKSTP\_OUT}}$ signal. Following are the state meaning and timing comments for the $\overline{\text{CKSTP\_OUT}}$ signal.

State Meaning Asserted—Indicates that a checkstop condition has been detected and the processor has ceased operation.

Negated—Indicates that the processor is operating normally.
See Chapter 8.

Timing Comments Assertion—May occur at any time and may be asserted asynchronously to input clocks.

Negation—Is negated upon assertion of $\overline{\text{HRESET}}$.

7.2.9.5 Reset Signals

There are two reset signals on Gekko—hard reset ($\overline{\text{HRESET}}$) and soft reset ($\overline{\text{SRESET}}$). Descriptions of the reset signals are as follows:

7.2.9.5.1 Hard Reset ($\overline{\text{HRESET}}$)—Input

The hard reset ($\overline{\text{HRESET}}$) signal must be used at power-on in conjunction with the $\overline{\text{TRST}}$ signal to properly reset the processor. Following are the state meaning and timing comments for the $\overline{\text{HRESET}}$ signal.

State Meaning Asserted—Initiates a complete hard reset operation when this input transitions from asserted to negated. Causes a reset exception as described in Section 4.5.1, “System Reset Exception (0x00100)" on Page 4-12 Output drivers are released to high impedance within five clocks after the assertion of $\overline{\text{HRESET}}$.

Negated—Indicates that normal operation should proceed. See Section 8.8.3, “Reset Inputs" on Page 8-37.

Timing Comments Assertion—May occur at any time and may be asserted asynchronously to Gekko input clock; must be held asserted for a minimum of 255 clock cycles after the PLL lock time has been met. Refer to Gekko hardware specifications for further timing comments.

Negation—May occur any time after the minimum reset pulse width has been met.

This input has additional functionality in certain test modes.

7.2.9.5.2 Soft Reset ($\overline{\text{SRESET}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{SRESET}}$ signal.

State Meaning Asserted— Initiates processing for a reset exception as described in Section 4.5.1, “System Reset Exception (0x00100)" on Page 4-12.

Negated—Indicates that normal operation should proceed. See Section 8.8.3, “Reset Inputs" on Page 8-37.

Timing Comments Assertion—May occur at any time and may be asserted asynchronously to Gekko input clock. The $\overline{\text{SRESET}}$ input is negative edge-sensitive.

Negation—May be negated two bus cycles after assertion.

This input has additional functionality in certain test modes.

7.2.9.6 Processor Status Signals

Processor status signals indicate the state of the processor. This includes the memory reservation signal, machine quiesce control signals, time base enable signal, and $\overline{\text{TLBISYNC}}$ signal.

7.2.9.6.1 Quiescent Request ($\overline{\text{QREQ}}$)—Output

Following are the state meaning and timing comments for $\overline{\text{QREQ}}$.

State Meaning Asserted—Indicates that Gekko is requesting all bus activity normally required to be snooped to terminate or to pause so Gekko may enter a quiescent (low power) state. When Gekko has entered a quiescent state, it no longer snoops bus activity.

Negated—Indicates that Gekko is not making a request to enter the quiescent state.

Timing Comments Assertion/Negation—May occur on any cycle. $\overline{\text{QREQ}}$ will remain asserted for the duration of the quiescent state.

7.2.9.6.2 Quiescent Acknowledge ($\overline{\text{QACK}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{QACK}}$ signal.

State Meaning Asserted—Indicates that all bus activity that requires snooping has terminated or paused, and that Gekko may enter the quiescent (or low power) state.

Negated—Indicates that Gekko may not enter a quiescent state, and must continue snooping the bus.

Timing Comments Assertion/Negation—May occur on any cycle following the assertion of $\overline{\text{QREQ}}$, and must be held asserted for at least one bus clock cycle.

Start-Up—$\overline{\text{QACK}}$ is sampled at the negation of $\overline{\text{HRESET}}$ to select 32-bit bus mode; if $\overline{\text{QACK}}$ is de-asserted at start-up, 32-bit bus mode is selected.

7.2.10 IEEE 1149.1a-1993 Interface Description

Gekko has five dedicated JTAG signals which are described in Table 7-6. The test data input (TDI) and test data output (TDO) scan ports are used to scan instructions as well as data into the various scan registers for JTAG operations. The scan operation is controlled by the

test access port (TAP) controller which in turn is controlled by the test mode select (TMS) input sequence. The scan data is latched in at the rising edge of test clock (TCK).

Table 7-6. IEEE Interface Pin Descriptions

Signal Name	Input/Output	Weak Pullup Provided	IEEE 1149.1a Function
TDI	Input	Yes	Serial scan input signal
TDO	Output	No	Serial scan output signal
TMS	Input	Yes	TAP controller mode signal
TCK	Input	Yes	Scan clock
TRST	Input	Yes	TAP controller reset

Test reset ($\overline{\text{TRST}}$) is a JTAG optional signal which is used to reset the TAP controller asynchronously. The $\overline{\text{TRST}}$ signal assures that the JTAG logic does not interfere with the normal operation of the chip, and must be asserted and deasserted coincident with the assertion of the $\overline{\text{HRESET}}$ signal.

7.2.11 Clock Signals

Gekko clock signal inputs determine the system clock frequency and provide a flexible clocking scheme that allows the processor to operate at an integer multiple of the system clock frequency.

Refer to Gekko hardware specifications for exact timing relationships of the clock signals.

7.2.11.1 System Clock (SYSCLK)—Input

Gekko requires a single system clock (SYSCLK) input. This input sets the frequency of operation for the bus interface. Internally, Gekko uses a phase-locked loop (PLL) circuit to generate a master clock for all of the CPU circuitry (including the bus interface circuitry) which is phase-locked to the SYSCLK input. The master clock may be set to an integer or half-integer multiple (2:1, 3:1, 3.5:1, 4:1, 4.5:1, 5:1, 5.5:1, 6:1, 6.5:1, 7:1, 7.5:1, 8:1 or 10:1) of the SYSCLK frequency allowing the CPU core to operate at an equal or greater frequency than the bus interface.

State Meaning Asserted/Negated—The SYSCLK input is the primary clock input for Gekko, and represents the bus clock frequency for Gekko bus operation. Internally, Gekko may be operating at an integer or half-integer multiple of the bus clock frequency.

Timing Comments Duty cycle—Refer to Gekko hardware specifications for timing comments.
Note: SYSCLK is used as the frequency reference for the internal PLL clock generator, and must not be suspended or varied during normal operation to ensure proper PLL operation.

7.2.11.2 Clock Out (CLK_OUT)—Output (N/A on Gekko)

The clock out (CLK_OUT) signal is an output signal (output-only). Following are the state meaning and timing comments for the CLK_OUT signal.

State Meaning Asserted/Negated—Provides PLL clock output for PLL testing and monitoring. The configuration of the HID0[SBCLK] and HID0[ECLK] bits determines whether the CLK_OUT signal clocks at either the processor clock frequency, the bus clock frequency, or half of the bus clock frequency. See Table 2-5 on Page 2-13 for HID0 register configuration of the CLK_OUT signal.
The CLK_OUT signal defaults to a high-impedance state following the assertion of $\overline{\text{HRESET}}$. The CLK_OUT signal is provided for testing only.

Timing Comments Assertion/Negation—Refer to Gekko hardware specifications for timing comments.

7.2.11.3 PLL Configuration (PLL_CFG[0–3])—Input

The PLL (phase-locked loop) is configured by the PLL_CFG[0–3] signals. For a given SYSCLK (bus) frequency, the PLL configuration signals set the internal CPU frequency of operation. Refer to Gekko hardware specifications for PLL configuration.

Following are the state meaning and timing comments for the PLL_CFG[0–3] signals.

State Meaning Asserted/Negated— Configures the operation of the PLL and the internal processor clock frequency. Settings are based on the desired bus and internal frequency of operation.

Timing Comments Assertion/Negation—Must remain stable during operation; should only be changed during the assertion of $\overline{\text{HRESET}}$ or during sleep mode. These bits may be read through the PC[0–3] bits in the HID1 register.

7.2.12 Power and Ground Signals

Gekko provides the following connections for power and ground:

- V_{DD}—The V_{DD} signals provide the supply voltage connection for the processor core.
- OV_{DD}—The OV_{DD} signals provide the supply voltage connection for the system interface drivers.
- AV_{DD}—The AV_{DD} power signal provides power to the clock generation phase-locked loop. See Gekko hardware specifications for information on how to use this signal.
- GND and OGND—The GND and OGND signals provide the connection for grounding Gekko. On Gekko, there is no electrical distinction between the GND and OGND signals.

Chapter 8 Bus Interface Operation

This chapter describes the Gekko microprocessor bus interface and its operation. It shows how the Gekko signals, defined in Chapter 7, "Signal Descriptions" interact to perform address and data transfers.

The bus interface buffers bus requests from the instruction and data caches, and executes the requests per the 60x bus protocol. It includes address register queues, prioritizing logic, and bus control logic. It captures snoop addresses for snooping in the cache and in the address register queues. It also snoops for reservations and holds the touch load address for the cache. All data storage for the address register buffers (load and store data buffers) are located in the cache section. The data buffers are considered temporary storage for the cache and not part of the bus interface.

The general functions and features of the bus interface are as follows:

- Seven address register buffers that include the following:
  - — Instruction cache load address buffer
  - — DMA load address buffer
  - — Data cache load address buffer (shared with DMA load)
  - — Two data cache castout/store address buffers (shared with write gather pipe)
  - — Data cache snoop copy-back address buffer (associated data block buffer located in cache)
  - — Reservation address buffer for snoop monitoring
- Pipeline collision detection for data cache buffers
- Reservation address snooping for **lwarx**/**stwcx.** instructions
- One-level address pipelining
- Load ahead of store capability

A conceptual block diagram of the bus interface is shown in Figure 8-1 on Page 8-2. The address register queues in the figure hold transaction requests that the bus interface may issue on the bus independently of the other requests. The bus interface may have up to two transactions operating on the bus at any given time through the use of address pipelining.

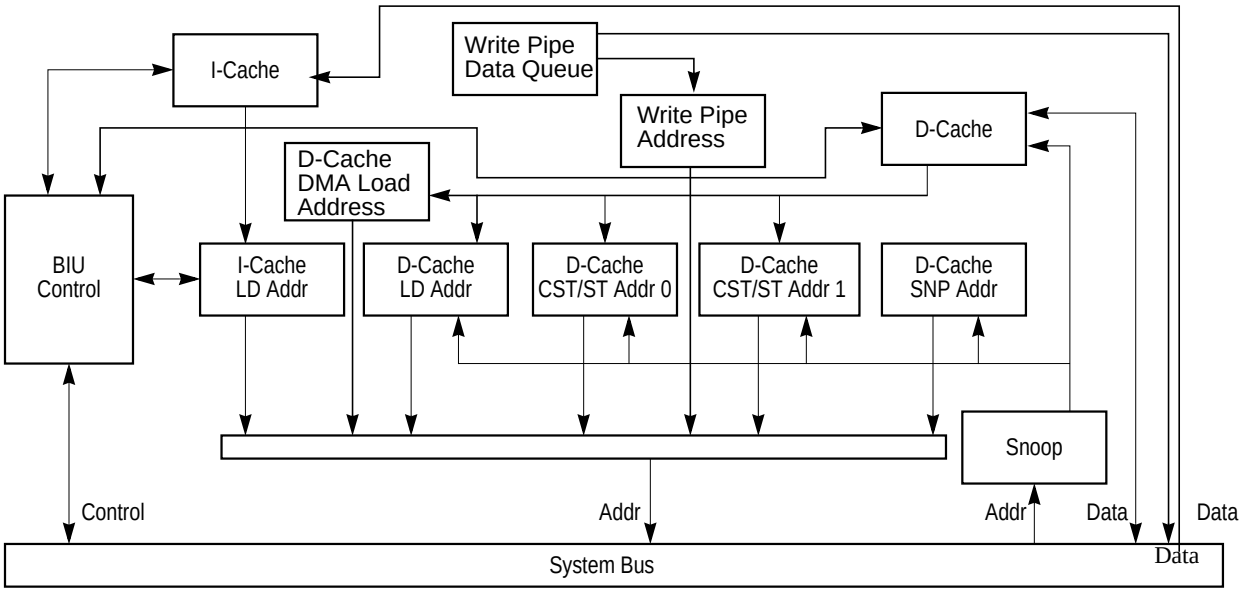


Figure 8-1. Bus Interface Address Buffers

8.1 Bus Interface Overview

The bus interface prioritizes requests for bus operations from the instruction and data caches, and performs bus operations in accordance with the protocol described in the *PowerPC Microprocessor Family: The Bus Interface for 32-Bit Microprocessors*. It includes address register queues, prioritization logic, and bus control unit. The bus interface latches snoop addresses for snooping in the data cache and in the address register queues, and for reservations controlled by the Load Word and Reserve Indexed (**lwarx**) and Store Word Conditional Indexed (**stwcx.**) instructions, and maintains the touch load address for the cache. The interface allows one level of pipelining; that is, with certain restrictions discussed later, there can be two outstanding transactions at any given time. Accesses are prioritized with load operations preceding store operations.

Instructions are automatically fetched from the memory system into the instruction unit where they are dispatched to the execution units at a peak rate of two instructions per clock. Conversely, load and store instructions explicitly specify the movement of operands to and from the integer and floating-point register files and the memory system.

When Gekko encounters an instruction or data access, it calculates the logical address (effective address in the architecture specification) and uses the low-order address bits to check for a hit in the on-chip, 32-Kbyte instruction and data caches.

During cache lookup, the instruction and data memory management units (MMUs) use the higher-order address bits to calculate the virtual address from which they calculate the physical address (real address in the architecture specification). The physical address bits are then compared with the corresponding cache tag bits to determine if a cache hit occurred in the L1 instruction or data cache. If the access misses in the corresponding cache, the physical address is used to access the L2 cache tags (if the L2 cache is enabled). If no match is found in the L2 cache tags, the physical address is used to access system memory.

In addition to the loads, stores, and instruction fetches, Gekko performs hardware table search

operations following TLB misses, L2 cache cast-out operations when least-recently used cache lines are written to memory after a cache miss, and cache-line snoop push-out operations when a modified cache line experiences a snoop hit from another bus master.

Figure 8-2 on Page 8-4 shows the address path from the execution units and instruction fetcher, through the translation logic to the caches and bus interface logic.

Gekko uses separate address and data buses and a variety of control and status signals for performing reads and writes. The address bus is 32 bits wide and the data bus is 64 bits wide. The interface is synchronous—all Gekko inputs are sampled at and all outputs are driven from the rising edge of the bus clock. The processor runs at a multiple of the bus-clock speed.

8.1.1 Operation of the Instruction and Data L1 Caches

Gekko provides independent instruction and data L1 caches. Each cache is a physically-addressed, 32-Kbyte cache with eight-way set associativity. Both caches consist of 128 sets of eight cache lines, with eight words in each cache line.

Because the data cache on Gekko is an on-chip, write-back primary cache, the predominant type of transaction for most applications is burst-read memory operations, followed by burst-write memory operations and single-beat (noncacheable or write-through) memory read and write operations. Additionally, there can be address-only operations, variants of the burst and single-beat operations (global memory operations that are snooped, and atomic memory operations, for example), and address retry activity (for example, when a snooped read access hits a modified line in the cache).

Since Gekko's data cache tags are single ported, simultaneous load or store, DMA access, and snoop accesses cause resource contention. Snoop accesses have the highest priority and are given first access to the tags, unless the snoop access coincides with a tag write, in which case the snoop is retried and must re-arbitrate for access to the cache. Loads or stores that are deferred due to snoop accesses are performed on the clock cycle following the snoop. DMA access has the lowest priority.

Gekko supports a three-state coherency protocol that supports the modified, exclusive, and invalid (MEI) cache states. The protocol is a subset of the MESI (modified/exclusive/shared/invalid) four-state protocol and operates coherently in systems that contain four-state caches.

With the exception of the **dcbz** instruction (and the **dcbi**, **dcbst**, and **dcbf** instructions, if HID0[ABE] is enabled), Gekko does not broadcast cache control instructions. The cache control instructions are intended for the management of the local cache but not for other caches in the system.

Instruction cache lines in Gekko are loaded in four beats of 64 bits each. The burst load is performed as critical double word first. The critical double word is simultaneously written to the cache and forwarded to the instruction pre-fetch unit, thus minimizing stalls due to load delays. If subsequent loads follow in sequential order, the instructions will be forwarded to the requesting unit as the cache block is written.

Data cache lines in Gekko are loaded into the cache in one cycle for 256 bits. For cache line load due to the cache miss of a load instruction, the critical double word is simultaneously written to the 256 bit line fill buffer and forwarded to the requesting load/store unit. If subsequent loads follow in sequential order, the data will be forwarded to the load/store unit as the cache block is written into the cache. For DMA read and data cache cast out, it takes one cycle to read the data out of the cache.

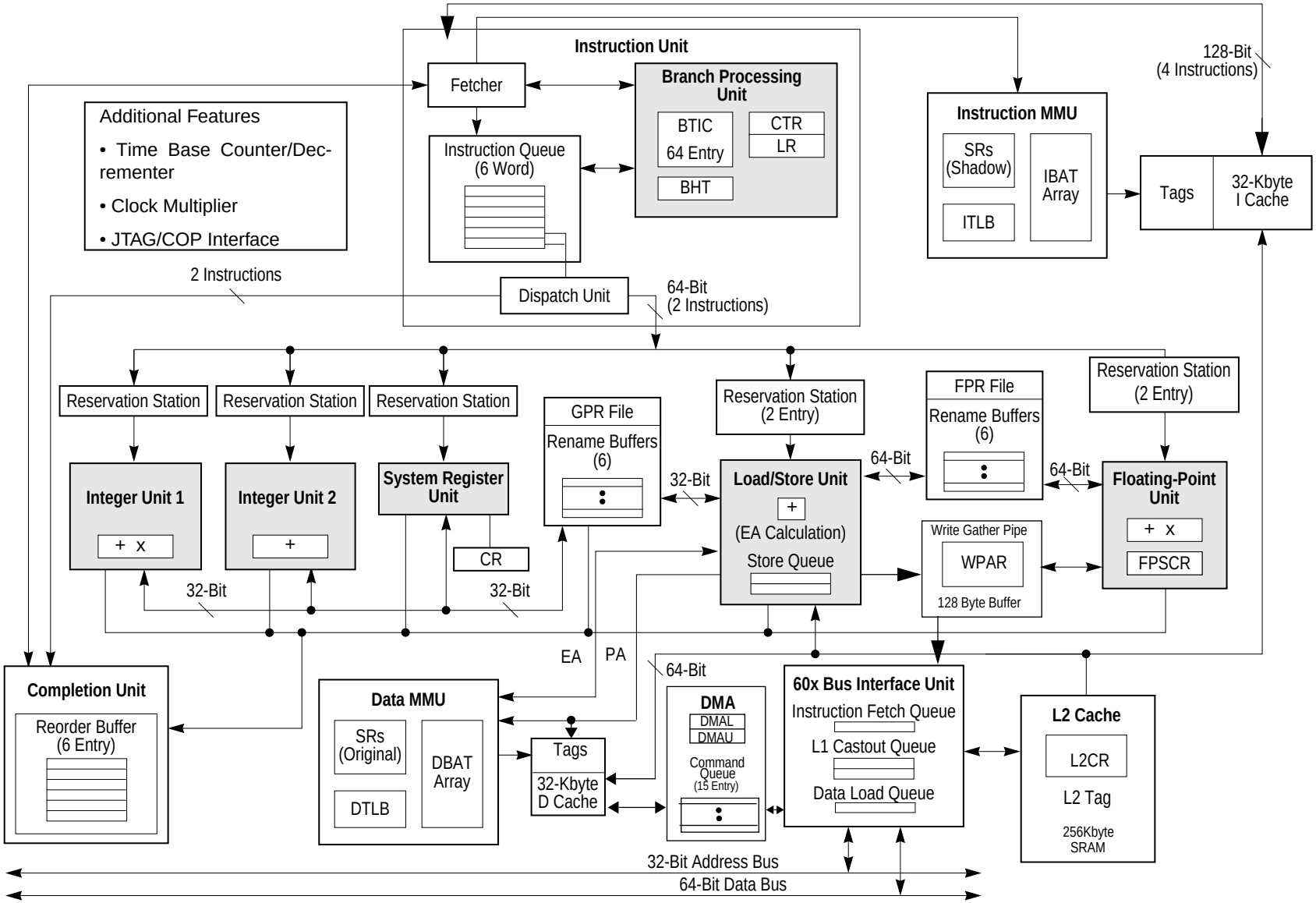


Figure 8-2. PowerPC Gekko Microprocessor Block Diagram

Cache lines are selected for replacement based on a pseudo least-recently-used (PLRU) algorithm. Each time a cache line is accessed, it is tagged as the most-recently-used line of the set. When a miss occurs, and all eight lines in the set are marked as valid, the least recently used line is replaced with the new data. When data to be replaced is in the modified state, the modified data is written into a write-back buffer while the missed data is being read from memory. When the load completes, Gekko then pushes the replaced line from the write-back buffer to the L2 cache (if enabled), or to main memory in a burst write operation.

8.1.2 Operation of the Bus Interface

Memory accesses can occur in single-beat (1, 2, 3, 4, and 8 bytes) and four-beat (32 bytes) burst data transfers. The address and data buses are independent for memory accesses to support pipelining and split transactions. Gekko can pipeline as many as two transactions and has limited support for out-of-order split-bus transactions.

Access to the bus interface is granted through an external arbitration mechanism that allows devices to compete for bus mastership. This arbitration mechanism is flexible, allowing Gekko to be integrated into systems that implement various fairness and bus-parking procedures to avoid arbitration overhead. Typically, memory accesses are weakly ordered to maximize the efficiency of the bus without sacrificing coherency of the data. Gekko allows load operations to bypass store operations (except when a dependency exists). In addition, Gekko can be configured to reorder high-priority store operations ahead of lower-priority store operations. Because the processor can dynamically optimize run-time ordering of load/store traffic, overall performance is improved.

NOTE: The synchronize (**sync**) and enforce in-order execution of IO (**eieio**) instructions can be used to enforce strong ordering.

The following sections describe how Gekko interface operates, providing detailed timing diagrams that illustrate how the signals interact. A collection of more general timing diagrams are included as examples of typical bus operations.

Figure 8-3 on Page 8-6 is a legend of the conventions used in the timing diagrams.

This is a synchronous interface—all Gekko input signals are sampled and output signals are driven on the rising edge of the bus clock cycle (see the *Gekko Datasheet* for exact timing information).

8.1.3 Direct-Store Accesses

Gekko does not support the extended transfer protocol for accesses to the direct-store storage space. The transfer protocol used for any given access is selected by the T bit in the MMU segment registers; if the T bit is set, the memory access is a direct-store access. An attempt to access instructions or data in a direct-store segment will result in Gekko taking an ISI or DSI exception.

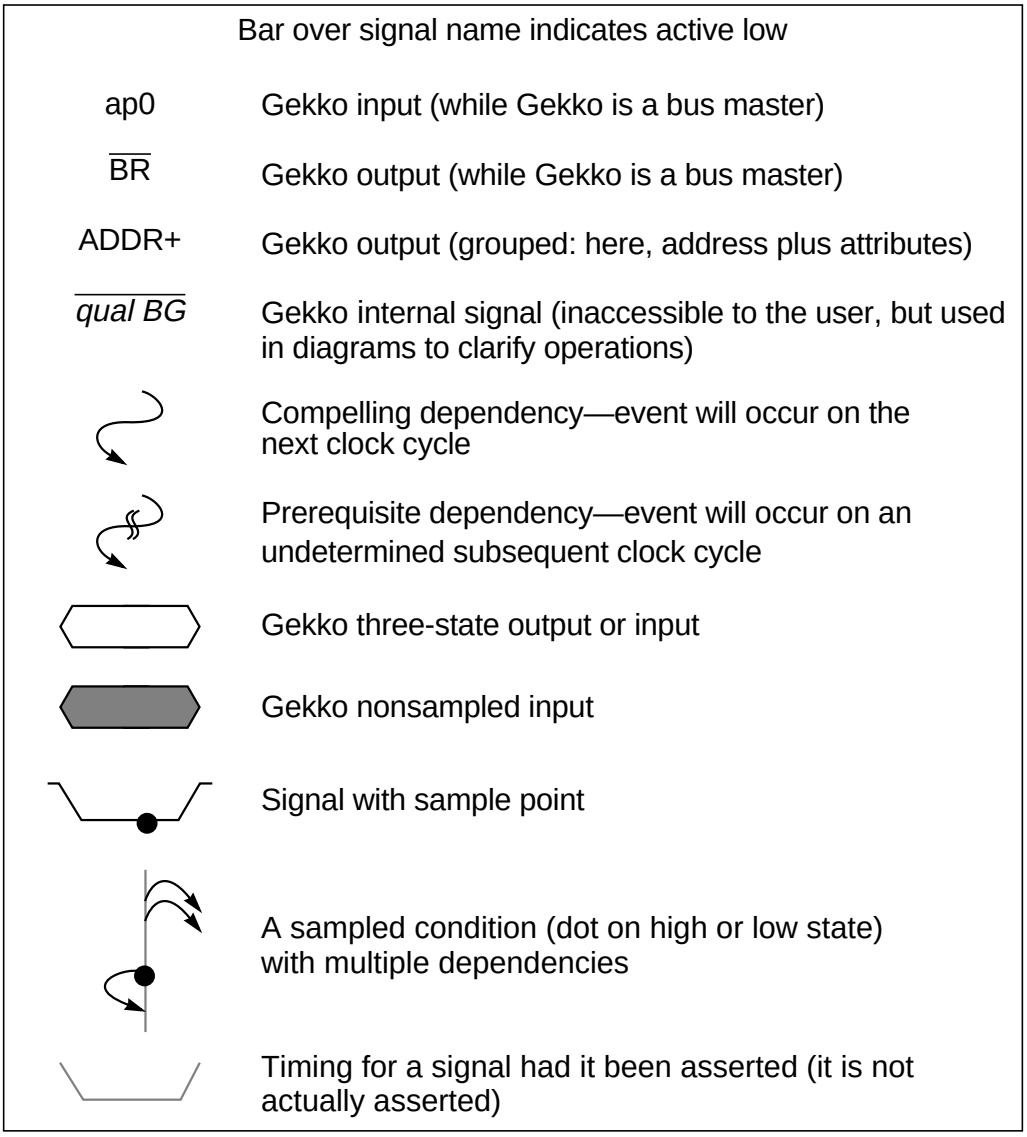


Figure 8-3. Timing Diagram Legend

8.2 Memory Access Protocol

Memory accesses are divided into address and data tenures. Each tenure has three phases—bus arbitration, transfer, and termination. Gekko also supports address-only transactions. Note that address and data tenures can overlap, as shown in Figure 8-4 on Page 8-7.

Figure 8-4 shows that the address and data tenures are distinct from one another and that both consist of three phases—arbitration, transfer, and termination. Address and data tenures are independent (indicated in Figure 8-4 by the fact that the data tenure begins before the address tenure ends), which allows split-bus transactions to be implemented at the system level in multiprocessor systems.

Figure 8-4 shows a data transfer that consists of a single-beat transfer of as many as 64 bits. Four-beat burst transfers of 32-byte cache lines require data transfer termination signals for each beat of data.

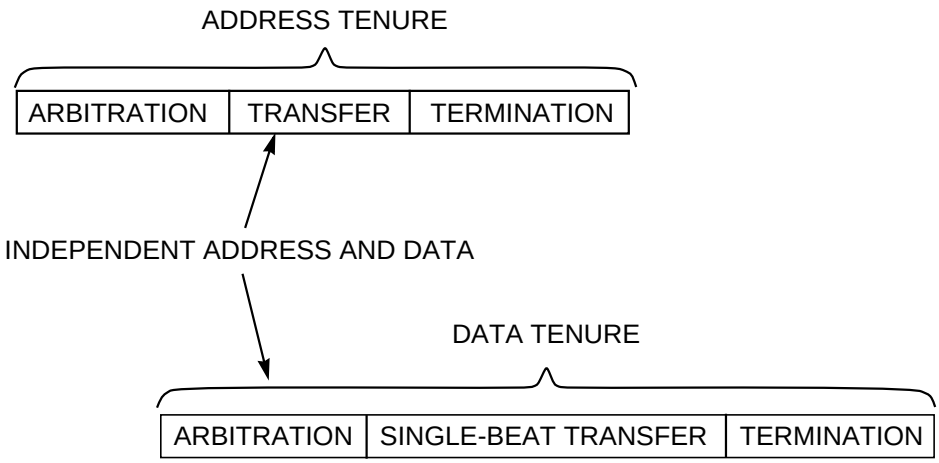


Figure 8-4. Overlapping Tenures on Gekko Bus for a Single-Beat Transfer

The basic functions of the address and data tenures are as follows:

- Address tenure
  - — Arbitration: During arbitration, address bus arbitration signals are used to gain mastership of the address bus.
  - — Transfer: After Gekko is the address bus master, it transfers the address on the address bus. The address signals and the transfer attribute signals control the address transfer. The address parity and address parity error signals ensure the integrity of the address transfer.
  - — Termination: After the address transfer, the system signals that the address tenure is complete or that it must be repeated.
- Data tenure
  - — Arbitration: To begin the data tenure, Gekko arbitrates for mastership of the data bus.
  - — Transfer: After Gekko is the data bus master, it samples the data bus for read operations or drives the data bus for write operations. The data parity and data parity error signals ensure the integrity of the data transfer.
  - — Termination: Data termination signals are required after each data beat in a data transfer. Note that in a single-beat transaction, the data termination signals also indicate the end of the tenure, while in burst accesses, the data termination signals apply to individual beats and indicate the end of the tenure only after the final data beat.

Gekko generates an address-only bus transfer during the execution of the **dcbz** instruction (and for the **dcbi**, **dcbf**, **dcbst**, **sync**, and **eieio** instructions, if HID0[ABE] is enabled), which uses only the address bus with no data transfer involved. Additionally, Gekko's retry capability provides an efficient snooping protocol for systems with multiple memory systems (including caches) that must remain coherent.

8.2.1 Arbitration Signals

Arbitration for both address and data bus mastership is performed by a central, external arbiter and, minimally, by the arbitration signals shown inSection 7.2.1, "Address Bus Arbitration Signals" on Page 7-3. Most arbiter implementations require additional signals to coordinate bus master/slave/snooping activities.

NOTE: Address bus busy ($\overline{ABB}$) and data bus busy ($\overline{DBB}$) signals are not supported on Gekko. Gekko uses internally generated signals, $\overline{iABB}$ and $\overline{iDBB}$ to determine the status of the bus transactions. Gekko does not support the $\overline{DRTRY}$ signal pin which is internally configured as a pull-up. All the references to the $\overline{DRTRY}$ signal shall be considered as a permanently negated signal.

The following list describes the address arbitration signals:

- **$\overline{BR}$ (bus request)**—Assertion indicates that Gekko is requesting mastership of the address bus.
- **$\overline{BG}$ (bus grant)**—Assertion indicates that Gekko may, with the proper qualification, assume mastership of the address bus. A qualified bus grant occurs when $\overline{BG}$ is asserted and when $\overline{iABB}$ and $\overline{ARTRY}$ are negated.

  If Gekko is parked, $\overline{BR}$ need not be asserted for the qualified bus grant.

The following list describes the data arbitration signals:

- **$\overline{DBG}$ (data bus grant**)—Indicates that Gekko may, with the proper qualification, assume mastership of the data bus. A qualified data bus grant occurs when $\overline{DBG}$ is asserted while $\overline{DRTRY}$, $\overline{iDBB}$ and $\overline{ARTRY}$ are negated.

  The $\overline{ARTRY}$ signal is driven from the bus and is only for the address bus tenure associated with the current data bus tenure (that is, not from another address tenure).

  Gekko always assumes data bus mastership if it needs the data bus and is given a qualified data bus grant.

  For more detailed information on the arbitration signals, refer to Section 7.2.1, "Address Bus Arbitration Signals" on Page 7-3 and Section 7.2.6, "Data Bus Arbitration Signals" on Page 7-12.

8.2.2 Address Pipelining and Split-Bus Transactions

Gekko protocol provides independent address and data bus capability to support pipelined and split-bus transaction system organizations. Address pipelining allows the address tenure of a new bus transaction to begin before the data tenure of the current transaction has finished. Split-bus transaction capability allows other bus activity to occur (either from the same master or from different masters) between the address and data tenures of a transaction.

While this capability does not inherently reduce memory latency, support for address pipelining and split-bus transactions can greatly improve effective bus/memory throughput. For this reason, these techniques are most effective in shared-memory multimaster implementations where bus bandwidth is an important measurement of system performance.

External arbitration is required in systems in which multiple devices must compete for the system bus. The design of the external arbiter affects pipelining by regulating address bus grant ($\overline{BG}$), data bus grant ($\overline{DBG}$), and address acknowledge ($\overline{AACK}$) signals. For example, a one-level pipeline is enabled by asserting $\overline{AACK}$ to the current address bus master and granting mastership of the address bus to the next requesting master before the current data bus tenure has completed. Two address tenures can

occur before the current data bus tenure completes.

Gekko can pipeline its own transactions to a depth of one level (intraprocessor pipelining); however, Gekko bus protocol does not constrain the maximum number of levels of pipelining that can occur on the bus between multiple masters (interprocessor pipelining). The external arbiter must control the pipeline depth and synchronization between masters and slaves.

In a pipelined implementation, data bus tenures are kept in strict order with respect to address tenures. However, external hardware can further decouple the address and data buses, allowing the data tenures to occur out of order with respect to the address tenures. This requires some form of system tag to associate the out-of-order data transaction with the proper originating address transaction (not defined for Gekko interface). Individual bus requests and data bus grants from each processor can be used by the system to implement tags to support interprocessor, out-of-order transactions.

NOTE: Gekko drops out of pipeline mode between consecutive burst data reads (with the exception of consecutive DMA reads, which are pipelined) and between consecutive burst instruction fetches. No other sequences of operations cause this effect. In this case, the address tenure of the second transaction will not begin until one to three bus clocks after the end of the data tenure of the first transaction.

8.3 Address Bus Tenure

This section describes the three phases of the address tenure—address bus arbitration, address transfer, and address termination.

8.3.1 Address Bus Arbitration

Gekko replaces the $\overline{\text{ABB}}$ signal with an internal signal, $\overline{\text{iABB}}$,which is asserted on $\overline{\text{TS}}$ and is negated the cycle after $\overline{\text{AACK}}$.

When Gekko needs access to the external bus and it is not parked ($\overline{\text{BG}}$ is negated), it asserts bus request ($\overline{\text{BR}}$) until it is granted mastership of the bus and the bus is available (see Figure 8-5 on Page 8-10). The external arbiter must grant master-elect status to the potential master by asserting the bus grant ($\overline{\text{BG}}$) signal. Gekko determines that the address bus is not busy by monitoring the $\overline{\text{TS}}$ and the $\overline{\text{AACK}}$ input signals. Gekko determines that the bus is available when the address bus is not busy, $\overline{\text{BG}}$ is asserted and the address retry ($\overline{\text{ARTRY}}$) input is negated. This is referred to as a qualified bus grant and Gekko can assume address bus mastership.

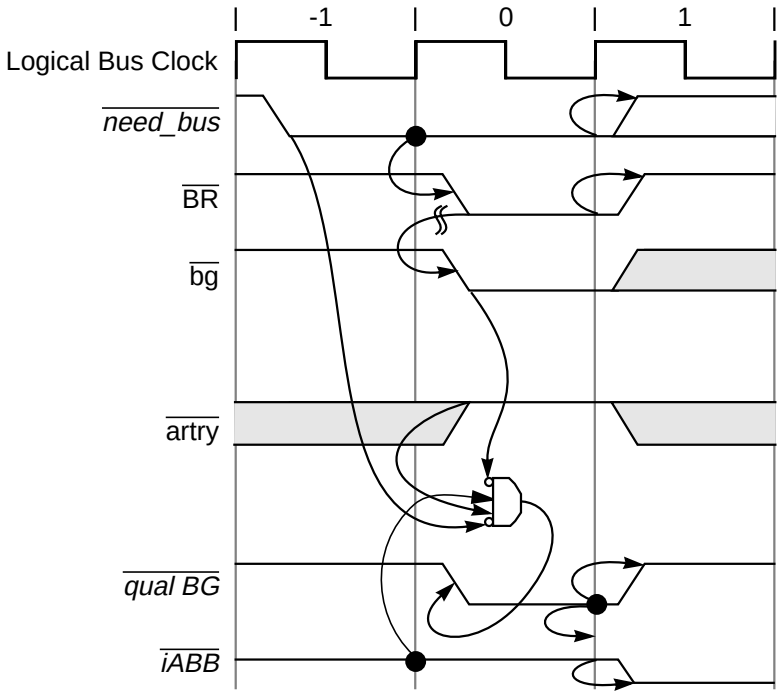


Figure 8-5. Address Bus Arbitration

External arbiters must allow only one device at a time to be the address bus master. For implementations in which no other device can be a master, $\overline{BG}$ can be grounded (always asserted) to continually grant mastership of the address bus to Gekko.

If Gekko asserts $\overline{BR}$ before the external arbiter asserts $\overline{BG}$, Gekko is considered to be unparked, as shown in Figure 8-5. Figure 8-6 on Page 8-11 shows the parked case, where a qualified bus grant exists on the clock edge following a need_bus condition. Notice that the bus clock cycle required for arbitration is eliminated if Gekko is parked, reducing overall memory latency for a transaction. Gekko always negates $\overline{iABB}$ for at least one bus clock cycle after $\overline{AACK}$ is asserted, even if it is parked and has another transaction pending.

Typically, bus parking is provided to the device that was the most recent bus master; however, system designers may choose other schemes such as providing unrequested bus grants in situations where it is easy to correctly predict the next device requesting bus mastership.

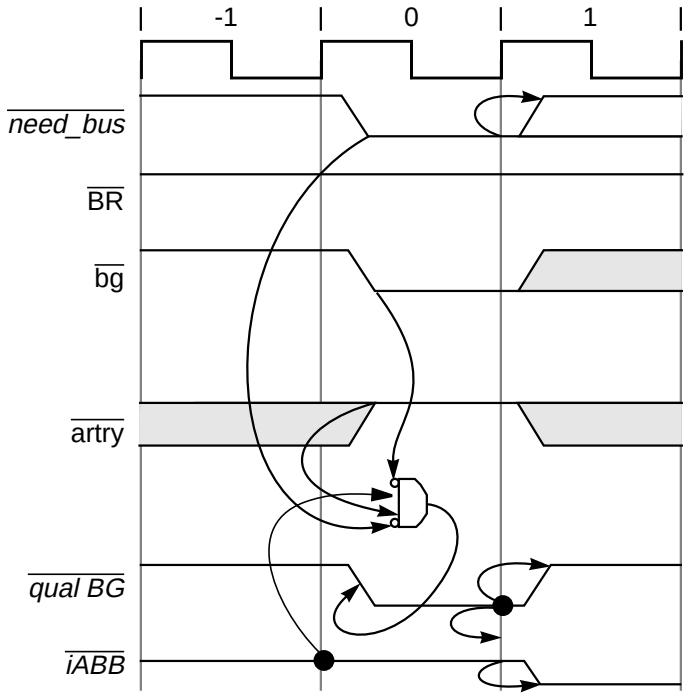


Figure 8-6. Address Bus Arbitration Showing Bus Parking

When Gekko receives a qualified bus grant, it assumes address bus mastership by negating the $\overline{BR}$ output signal. Meanwhile, Gekko drives the address for the requested access onto the address bus and asserts $\overline{TS}$ to indicate the start of a new transaction.

When designing external bus arbitration logic, note that Gekko may assert $\overline{BR}$ without using the bus after it receives the qualified bus grant. For example, in a system using bus snooping, if Gekko asserts $\overline{BR}$ to perform a replacement copy-back operation, another device can invalidate that line before Gekko is granted mastership of the bus. In these instances, Gekko asserts $\overline{BR}$ for at least one clock cycle.

System designers should note that Gekko does not support the $\overline{ABB}$ signal. The memory controller must monitor the $\overline{TS}$ and $\overline{AACK}$ input signals to determine the status of the address bus. Gekko allows this operation by using an internal version of $\overline{ABB}$ to determine if a qualified bus grant state exists.

Gekko will not qualify a bus grant during the cycle that $\overline{TS}$ is asserted on the bus by any master. Address bus arbitration requires that every assertion of $\overline{TS}$ be acknowledged by an assertion of $\overline{AACK}$ while the processor is not in sleep mode.

8.3.2 Address Transfer

During the address transfer, the physical address and all attributes of the transaction are transferred from the bus master to the slave device(s). Snooping logic may monitor the transfer to enforce cache coherency; see discussion about snooping in Section 8.3.3, "Address Transfer Termination" on Page 8-16. The signals used in the address transfer include the following signal groups:

- Address transfer start signal: transfer start ($\overline{TS}$)
- Address transfer signals: address bus (A[0–31]), and address parity (AP[0–3])
- Address transfer attribute signals: transfer type (TT[0–4]), transfer size (TSIZ[0–2]), transfer burst ($\overline{TBST}$), cache inhibit ($\overline{CI}$), write-through ($\overline{WT}$), and global ($\overline{GBL}$).

Figure 8-7 shows that the timing for all of these signals, except $\overline{TS}$, is identical. All of the address transfer and address transfer attribute signals are combined into the ADDR+ grouping in Figure 8-7. The $\overline{TS}$ signal indicates that Gekko has begun an address transfer and that the address and transfer attributes are valid (within the context of a synchronous bus).

In Figure 8-7, the address transfer occurs during bus clock cycles 1 and 2 (arbitration occurs in bus clock cycle 0 and the address transfer is terminated in bus clock 3). In this diagram, the address bus termination input, $\overline{AACK}$, is asserted to Gekko on the bus clock following assertion of $\overline{TS}$ (as shown by the dependency line). This is the minimum duration of the address transfer for Gekko; the duration can be extended by delaying the assertion of $\overline{AACK}$ for one or more bus clocks.

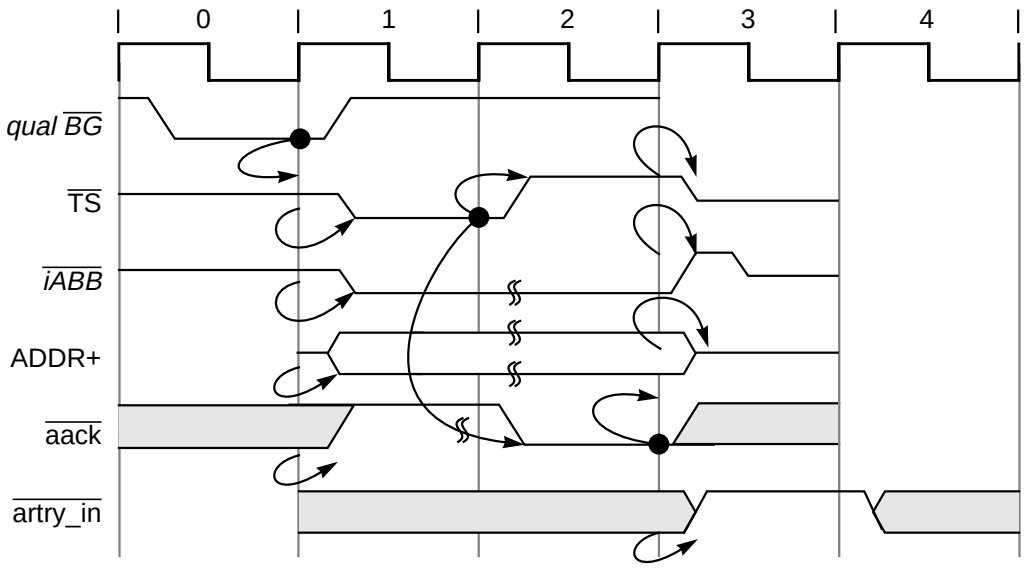


Figure 8-7. Address Bus Transfer

8.3.2.1 Address Bus Parity (N/A on Gekko)

Note: The address bus parity pins are not connected on Gekko, so it neither generates nor checks for correct parity on the external bus.

The BIU always generates 1 bit of correct odd-byte parity for each of the 4 bytes of address when a valid address is on the bus. The calculated values are placed on the AP[0–3] outputs when the BIU is the address bus master. If the BIU is not the master and $\overline{TS}$ and $\overline{GBL}$ are asserted together (qualified condition for snooping memory operations), the calculated values are compared with the AP[0–3] inputs. If there is an error, and address parity checking is enabled (HID0[EBA] set to 1), a machine check exception is generated. An address bus parity error causes a checkstop condition if MSR[ME] is cleared to 0. For more information about checkstop conditions, see Chapter 4, "Exceptions" in this manual.

8.3.2.2 Address Transfer Attribute Signals

The transfer attribute signals include several encoded signals such as the transfer type (TT[0–4]) signals, transfer burst ($\overline{TBST}$) signal, transfer size (TSIZ[0–2]) signals, write-through ($\overline{WT}$), and cache inhibit ($\overline{CI}$). Section 7.2.4, "Address Transfer Attribute Signals" on Page 7-6 describes the encodings for the address transfer attribute signals.

8.3.2.2.1 Transfer Type (TT[0–4]) Signals

Snooping logic should fully decode the transfer type signals if the $\overline{GBL}$ signal is asserted. Slave devices can sometimes use the individual transfer type signals without fully decoding the group. For a complete description of the encoding for TT[0–4], refer to Table 8-1 and Table 8-2 on Page 8-14.

8.3.2.2.2 Transfer Size (TSIZ[0–2]) Signals

The TSIZ[0–2] signals indicate the size of the requested data transfer as shown in Table 8-1. The TSIZ[0–2] signals may be used along with $\overline{TBST}$ and A[29–31] to determine which portion of the data bus contains valid data for a write transaction or which portion of the bus should contain valid data for a read transaction. Note that for a burst transaction (as indicated by the assertion of $\overline{TBST}$), TSIZ[0–2] are always set to 0b010. Therefore, if the $\overline{TBST}$ signal is asserted, the memory system should transfer a total of eight words (32 bytes), regardless of the TSIZ[0–2] encodings.

Table 8-1. Transfer Size Signal Encodings

$\overline{TBST}$	TSIZ0	TSIZ1	TSIZ2	Transfer Size
Asserted	0	1	0	Eight-word burst
Negated	0	0	0	Eight bytes
Negated	0	0	1	One byte
Negated	0	1	0	Two bytes
Negated	0	1	1	Three bytes
Negated	1	0	0	Four bytes
Negated	1	0	1	Five bytes (N/A)
Negated	1	1	0	Six bytes (N/A)
Negated	1	1	1	Seven bytes (N/A)

The basic coherency size of the bus is defined to be 32 bytes (corresponding to one cache line). Data transfers that cross an aligned, 32-byte boundary either must present a new address onto the bus at that boundary (for coherency consideration) or must operate as noncoherent data with respect to Gekko. Gekko never generates a bus transaction with a transfer size of 5 bytes, 6 bytes, or 7 bytes.

8.3.2.2.3 Write-Through ($\overline{WT}$) Signal

Gekko provides the $\overline{WT}$ signal to indicate a write-through operation as determined by the WIM bit settings during address translation by the MMU. The $\overline{WT}$ signal is also asserted for burst writes due to the execution of the **dcbf** and **dcbst** instructions, and snoop push operations. The $\overline{WT}$ signal is deasserted for accesses caused by the execution of the **ecowx** instruction. During read operations Gekko uses the $\overline{WT}$ signal to indicate whether the transaction is an instruction fetch ($\overline{WT}$ set to 1), or a data read operation ($\overline{WT}$ cleared to 0).

8.3.2.2.4 Cache Inhibit ($\overline{CI}$) Signal

Gekko indicates the caching-inhibited status of a transaction (determined by the setting of the WIM bits by the MMU) through the use of the $\overline{CI}$ signal. The $\overline{CI}$ signal is asserted even if the L1 caches are disabled or locked. This signal is also asserted for bus transactions caused by the execution of **eciwx** and **ecowx** instructions independent of the address translation.

8.3.2.3 Burst Ordering During Data Transfers

During burst data transfer operations, 32 bytes of data (one cache line) are transferred to or from the cache in order. Burst write transfers are always performed zero double word first, but since burst reads are performed critical double word first, a burst read transfer may not start with the first double word of the cache line, and the cache line fill may wrap around the end of the cache line.

Table 8-2 describes the data bus burst ordering.

Table 8-2. Burst Ordering

Data Transfer	For Starting Address:			
	A[27–28] = 00	A[27–28] = 01	A[27–28] = 10	A[27–28] = 11
First data beat	DW0	DW1	DW2	DW3
Second data beat	DW1	DW2	DW3	DW0
Third data beat	DW2	DW3	DW0	DW1
Fourth data beat	DW3	DW0	DW1	DW2

Note: A[29–31] are always 0b000 for burst transfers by Gekko.

8.3.2.4 Effect of Alignment in Data Transfers

Table 8-3 lists the aligned transfers that can occur on the Gekko bus. These are transfers in which the data is aligned to an address that is an integral multiple of the size of the data. For example, Table 8-3 shows that 1-byte data is always aligned; however, for a 4-byte word to be aligned, it must be oriented on an address that is a multiple of 4.

Table 8-3. Aligned Data Transfers

Transfer Size	TSIZ0	TSIZ1	TSIZ2	A[29–31]	Data Bus Byte Lane(s)							
					0	1	2	3	4	5	6	7
Byte	0	0	1	000	x	—	—	—	—	—	—	—
	0	0	1	001	—	x	—	—	—	—	—	—
	0	0	1	010	—	—	x	—	—	—	—	—
	0	0	1	011	—	—	—	x	—	—	—	—
	0	0	1	100	—	—	—	—	x	—	—	—
	0	0	1	101	—	—	—	—	—	x	—	—
	0	0	1	110	—	—	—	—	—	—	x	—
	0	0	1	111	—	—	—	—	—	—	—	x
Half word	0	1	0	000	x	x	—	—	—	—	—	—
	0	1	0	010	—	—	x	x	—	—	—	—
	0	1	0	100	—	—	—	—	x	x	—	—
	0	1	0	110	—	—	—	—	—	—	x	x
Word	1	0	0	000	x	x	x	x	—	—	—	—
	1	0	0	100	—	—	—	—	x	x	x	x
Double word	0	0	0	000	x	x	x	x	x	x	x	x

Note: The entries with an “x” indicate the byte portions of the requested operand which are read or written during a bus transaction.
The entries with a “–” are not required and are ignored during read transactions, and they are driven with undefined data during all write transactions.

Gekko supports misaligned memory operations, although their use may substantially degrade performance. Misaligned memory transfers address memory that is not aligned to the size of the data being transferred (such as, a word read of an odd byte address). Although most of these operations hit in the primary cache (or generate burst memory operations if they miss), Gekko interface supports misaligned transfers within a word (32-bit aligned) boundary, as shown in Table 8-4 on Page 8-16.

NOTE: The 4-byte transfer in Table 8-4 is only one example of misalignment. As long as the attempted transfer does not cross a word boundary, Gekko can transfer the data on the misaligned address (for example, a half-word read from an odd byte-aligned address). An attempt to address data that crosses a word boundary requires two bus transfers to access the data.

Due to the performance degradations associated with misaligned memory operations, they are best avoided. In addition to the double-word straddle boundary condition, the address translation logic can generate substantial exception overhead when the load/store multiple and load/store string instructions access misaligned data. It is strongly recommended that software attempt to align data where possible.

Table 8-4. Misaligned Data Transfers (Four-Byte Examples)

Transfer Size (Four Bytes)	TSIZ[0–2]	A[29–31]	Data Bus Byte Lanes							
			0	1	2	3	4	5	6	7
Aligned	1 0 0	0 0 0	A	A	A	A	—	—	—	—
Misaligned—first access	0 1 1	0 0 1		A	A	A	—	—	—	—
second access	0 0 1	1 0 0	—	—	—	—	A	—	—	—
Misaligned—first access	0 1 0	0 1 0	—	—	A	A	—	—	—	—
second access	0 1 1	1 0 0	—	—	—	—	A	A	—	—
Misaligned—first access	0 0 1	0 1 1	—	—	—	A	—	—	—	—
second access	0 1 1	1 0 0	—	—	—	—	A	A	A	—
Aligned	1 0 0	1 0 0	—	—	—	—	A	A	A	A
Misaligned—first access	0 1 1	1 0 1	—	—	—	—	—	A	A	A
second access	0 0 1	0 0 0	A	—	—	—	—	—	—	—
Misaligned—first access	0 1 0	1 1 0	—	—	—	—	—	—	A	A
second access	0 1 0	0 0 0	A	A	—	—	—	—	—	—
Misaligned—first access	0 0 1	1 1 1	—	—	—	—	—	—	—	A
second access	0 1 1	0 0 0	A	A	A	—	—	—	—	—

Notes:

A: Byte lane used
—: Byte lane not used

8.3.2.5 Alignment of External Control Instructions

The size of the data transfer associated with the **eciwx** and **ecowx** instructions is always 4 bytes. If the **eciwx** or **ecowx** instruction is misaligned and crosses any word boundary, Gekko will generate an alignment exception.

8.3.3 Address Transfer Termination

The address tenure of a bus operation is terminated when completed with the assertion of $\overline{\text{AACK}}$, or retried with the assertion of $\overline{\text{ARTRY}}$. Gekko does not terminate the address transfer until the $\overline{\text{AACK}}$ (address acknowledge) input is asserted; therefore, the system can extend the address transfer phase by delaying the assertion of $\overline{\text{AACK}}$ to Gekko. The assertion of $\overline{\text{AACK}}$ can be as early as the bus clock cycle following $\overline{\text{TS}}$ (see Figure 8-8 on Page 8-18), which allows a minimum address tenure of two

bus cycles. As shown in Figure 8-8, these signals are asserted for one bus clock cycle, three-stated for half of the next bus clock cycle, driven high till the following bus cycle, and finally three-stated. Note that $\overline{\text{AACK}}$ must be asserted for only one bus clock cycle.

The address transfer can be terminated with the requirement to retry if $\overline{\text{ARTRY}}$ is asserted anytime during the address tenure and through the cycle following $\overline{\text{AACK}}$. The assertion causes the entire transaction (address and data tenure) to be rerun. As a snooping device, Gekko asserts $\overline{\text{ARTRY}}$ for a snooped transaction that hits modified data in the data cache that must be written back to memory, or if the snooped transaction could not be serviced. As a bus master, Gekko responds to an assertion of $\overline{\text{ARTRY}}$ by aborting the bus transaction and re-requesting the bus. Note that after recognizing an assertion of $\overline{\text{ARTRY}}$ and aborting the transaction in progress, Gekko is not guaranteed to run the same transaction the next time it is granted the bus due to internal reordering of load and store operations.

If an address retry is required, the $\overline{\text{ARTRY}}$ response will be asserted by a bus snooping device as early as the second cycle after the assertion of $\overline{\text{TS}}$. Once asserted, $\overline{\text{ARTRY}}$ must remain asserted through the cycle after the assertion of $\overline{\text{AACK}}$. The assertion of $\overline{\text{ARTRY}}$ during the cycle after the assertion of $\overline{\text{AACK}}$ is referred to as a qualified $\overline{\text{ARTRY}}$. An earlier assertion of $\overline{\text{ARTRY}}$ during the address tenure is referred to as an early $\overline{\text{ARTRY}}$.

As a bus master, Gekko recognizes either an early or qualified $\overline{\text{ARTRY}}$ and prevents the data tenure associated with the retried address tenure. If the data tenure has already begun, Gekko aborts and terminates the data tenure immediately even if the burst data has been received. If the assertion of $\overline{\text{ARTRY}}$ is received up to or on the bus cycle following the first (or only) assertion of $\overline{\text{TA}}$ for the data tenure, Gekko ignores the first data beat, and if it is a load operation, does not forward data internally to the cache and execution units. If $\overline{\text{ARTRY}}$ is asserted after the first (or only) assertion of $\overline{\text{TA}}$, improper operation of the bus interface may result.

During the clock of a qualified $\overline{\text{ARTRY}}$, Gekko also determines if it should negate $\overline{\text{BR}}$ and ignore $\overline{\text{BG}}$ on the following cycle. On the following cycle, only the snooping master that asserted $\overline{\text{ARTRY}}$ and needs to perform a snoop copy-back operation is allowed to assert $\overline{\text{BR}}$. This guarantees the snooping master an opportunity to request and be granted the bus before the just-retried master can restart its transaction. Note that a nonclocked bus arbiter may detect the assertion of address bus request by the bus master that asserted $\overline{\text{ARTRY}}$, and return a qualified bus grant one cycle earlier than shown in Figure 8-8 on Page 8-18.

Note that if Gekko asserts $\overline{\text{ARTRY}}$ due to a snoop operation, and asserts $\overline{\text{BR}}$ in the bus cycle following $\overline{\text{ARTRY}}$ in order to perform a snoop push to memory it may be several bus cycles later before Gekko will be able to accept a $\overline{\text{BG}}$. (The delay in responding to the assertion of $\overline{\text{BG}}$ only occurs during snoop pushes from the L2 cache.) The bus arbiter should keep $\overline{\text{BG}}$ asserted until it detects $\overline{\text{BR}}$ negated or $\overline{\text{TS}}$ asserted from Gekko indicating that the snoop copy-back has begun. The system should ensure that no other address tenures occur until the current snoop push from Gekko is completed. Snoop push delays can also be avoided by operating the L2 cache in write-through mode so no snoop pushes are required by the L2 cache.

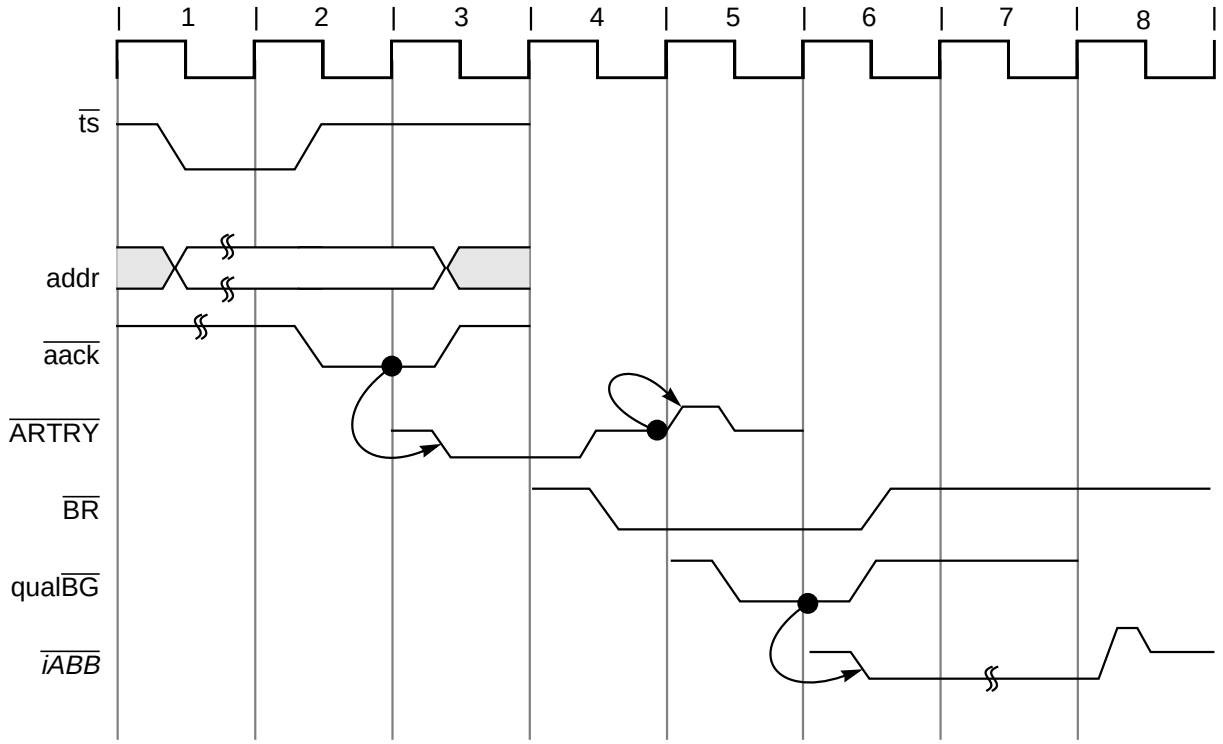


Figure 8-8. Snooped Address Cycle with ARTRY

8.4 Data Bus Tenure

This section describes the data bus arbitration, transfer, and termination phases defined by Gekko memory access protocol. The phases of the data tenure are identical to those of the address tenure, underscoring the symmetry in the control of the two buses.

Gekko does not support the DBB signal, typically found on a 60x PowerPC processor. Instead, Gekko uses an internal signal, iDBB . The iDBB signal is asserted on the bus clock cycle following a qualified DBG and is negated at least one bus clock cycle after the assertion of the final TA. Also, Gekko is configured to operate in no-DRTRY mode, so the state of the DRTRY signal as described in the following sections is ignored by the processor.

8.4.1 Data Bus Arbitration

Data bus arbitration uses the data arbitration signal group—DBG and iDBB. Additionally, the combination of TS and TT[0–4] provides information about the data bus request to external logic.

The TS signal is an implied data bus request from Gekko. The arbiter must qualify TS with the transfer type (TT) encodings to determine if the current address transfer is an address-only operation, which does not require a data bus transfer. If the data bus is needed, the arbiter grants data bus mastership by asserting the DBG input to Gekko. As with the address bus arbitration phase, Gekko must qualify the DBG input with a number of input signals before assuming bus mastership, as shown in Figure 8-9 on Page 8-19.

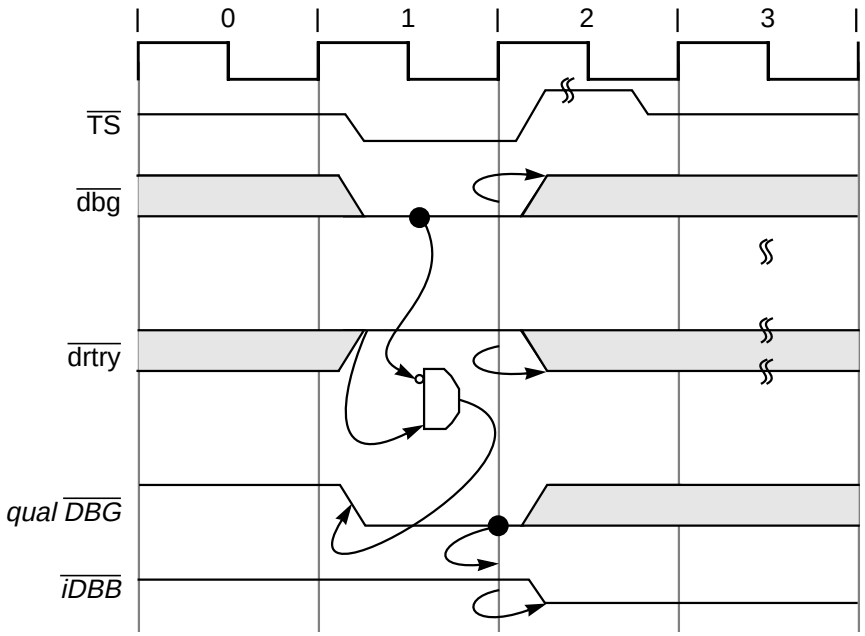


Figure 8-9. Data Bus Arbitration

A qualified data bus grant can be expressed as the following:

> QDBG = $\overline{\text{DBG}}$ asserted while $\overline{\text{DRTRY}}$, $\overline{\text{iDBB}}$ and $\overline{\text{ARTRY}}$ (associated with the data bus operation) are negated.

When a data tenure overlaps with its associated address tenure, a qualified $\overline{\text{ARTRY}}$ assertion coincident with a data bus grant signal does not result in data bus mastership Otherwise, Gekko always becomes the bus master on the bus clock cycle after recognition of a qualified data bus grant. Since Gekko can pipeline transactions, there may be an outstanding data bus transaction when a new address transaction is retried. In this case, Gekko becomes the data bus master to complete the outstanding transaction.

Gekko does not support the $\overline{\text{DBB}}$ signal. The memory system must track the start and end of the data tenure and control data tenure scheduling directly with $\overline{\text{DBG}}$. The $\overline{\text{DBG}}$ signal is only asserted to the next bus master the cycle before the cycle that the next bus master may actually begin its data tenure. Gekko always requires one cycle after data tenure completion before recognizing a qualified data bus grant for another data tenure.

8.4.2 Data Transfer

The data transfer signals include DH[0–31], DL[0–31], and DP[0–7]. For memory accesses, the DH and DL signals form a 64-bit data path for read and write operations.

Gekko transfers data in either single- or four-beat burst transfers. Single-beat operations can transfer from 1 to 8 bytes at a time and can be misaligned; see Section 8.3.2.4, "Effect of Alignment in Data Transfers" on Page 8-15. Burst operations always transfer eight words and are aligned on eight-word address boundaries. Burst transfers can achieve significantly higher bus throughput than single-beat operations.

The type of transaction initiated by Gekko depends on whether the code or data is cacheable and, for store operations whether the cache is in write-back or write-through mode, which software controls on either a page or block basis. Burst transfers support cacheable operations only; that is, memory structures must be marked as cacheable (and write-back for data store operations) in the

respective page or block descriptor to take advantage of burst transfers.

The Gekko output $\overline{\text{TBST}}$ indicates to the system whether the current transaction is a single- or four-beat transfer (except during **eciwx**/**ecowx** transactions, when it signals the state of EAR[28]). A burst transfer has an assumed address order. For load or store operations that miss in the cache (and are marked as cacheable and, for stores, write-back in the MMU), Gekko uses the double-word-aligned address associated with the critical code or data that initiated the transaction. This minimizes latency by allowing the critical code or data to be forwarded to the processor before the rest of the cache line is filled. For all other burst operations, however, the cache line is transferred beginning with the eight-word-aligned data.

8.4.3 Data Transfer Termination

Three signals are used to terminate data bus transactions—$\overline{\text{TA}}$, $\overline{\text{TEA}}$ (transfer error acknowledge), and $\overline{\text{ARTRY}}$.

The $\overline{\text{TA}}$ signal indicates normal termination of data transactions. It must always be asserted on the bus cycle coincident with the data that it is qualifying. It may be withheld by the slave for any number of clocks until valid data is ready to be supplied or accepted. Upon receiving a final (or only) termination condition, Gekko always negates $\overline{\text{iDBB}}$ for one cycle.

The $\overline{\text{TEA}}$ signal is used to signal a nonrecoverable error during the data transaction. It may be asserted on any cycle during a data bus tenure. The assertion of $\overline{\text{TEA}}$ terminates the data tenure immediately even if in the middle of a burst; however, it does not prevent incorrect data that has just been acknowledged with $\overline{\text{TA}}$ from being written into Gekko's cache or GPRs. The assertion of $\overline{\text{TEA}}$ initiates either a machine check exception or a checkstop condition based on the setting of the MSR[ME] bit.

An assertion of $\overline{\text{ARTRY}}$ causes the data tenure to be terminated immediately if the $\overline{\text{ARTRY}}$ is for the address tenure associated with the data tenure in operation. If $\overline{\text{ARTRY}}$ is connected for Gekko, the earliest allowable assertion of $\overline{\text{TA}}$ to Gekko is directly dependent on the earliest possible assertion of $\overline{\text{ARTRY}}$ to Gekko; see Section 8.3.3, "Address Transfer Termination" on Page 8-16.

8.4.3.1 Normal Single-Beat Termination

Normal termination of a single-beat data read operation occurs when $\overline{TA}$ is asserted by a responding slave. The $\overline{TEA}$ and $\overline{DRTRY}$ signals must remain negated during the transfer (see Figure 8-10).

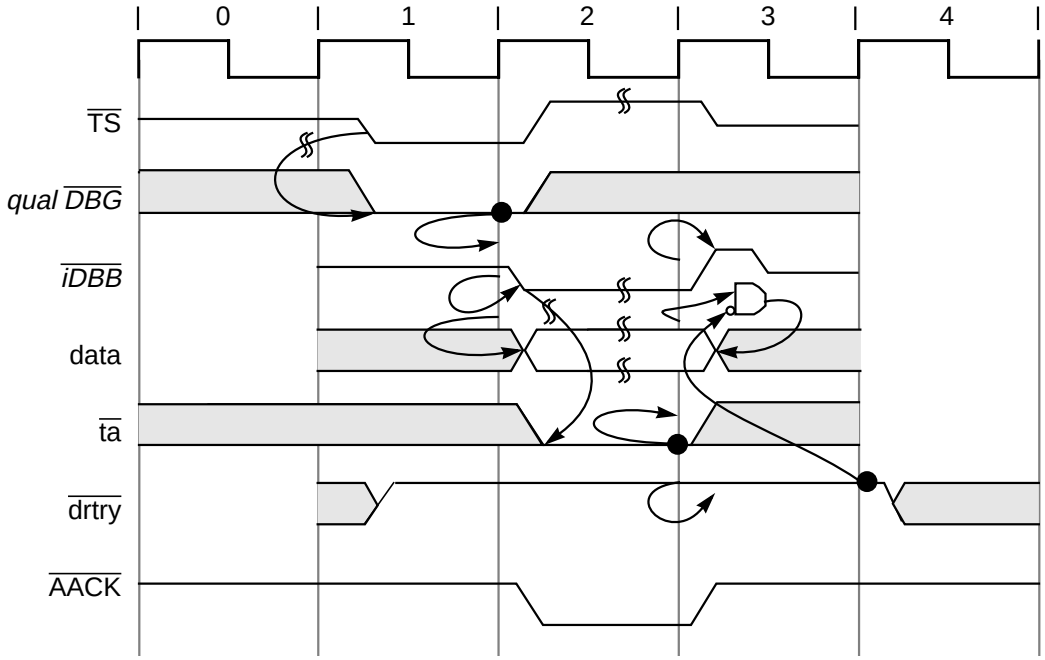


Figure 8-10. Normal Single-Beat Read Termination

The $\overline{DRTRY}$ signal is not sampled during data writes, as shown in Figure 8-11.

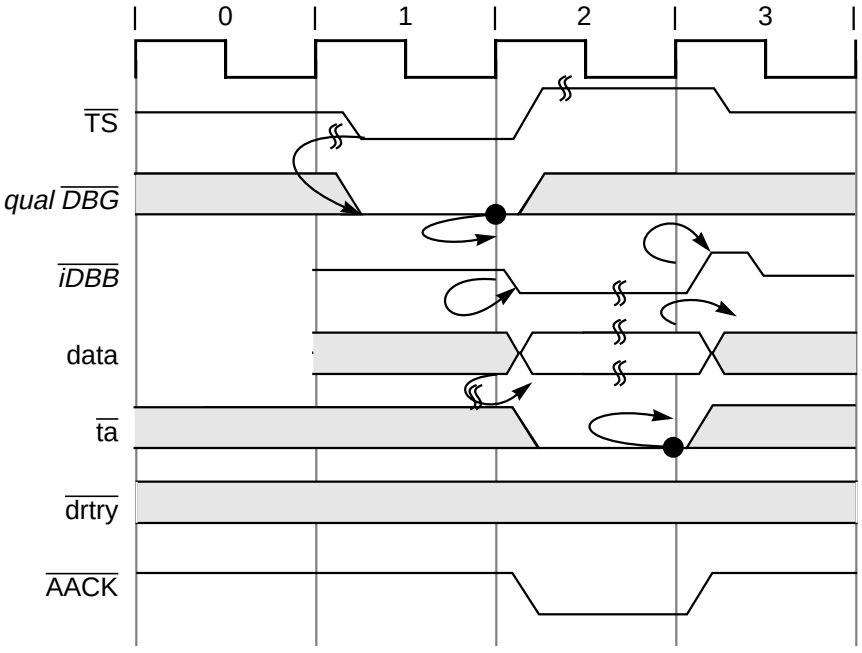


Figure 8-11. Normal Single-Beat Write Termination

Normal termination of a burst transfer occurs when $\overline{TA}$ is asserted for four bus clock cycles, as shown inFigure 8-12 on Page 8-22. The bus clock cycles in which $\overline{TA}$ is asserted need not be consecutive, thus allowing pacing of the data transfer beats. For read bursts to terminate successfully, $\overline{TEA}$ and $\overline{DRTRY}$ must remain negated during the transfer. For write bursts, $\overline{TEA}$ must remain negated for a successful transfer. $\overline{DRTRY}$ is ignored during data writes.

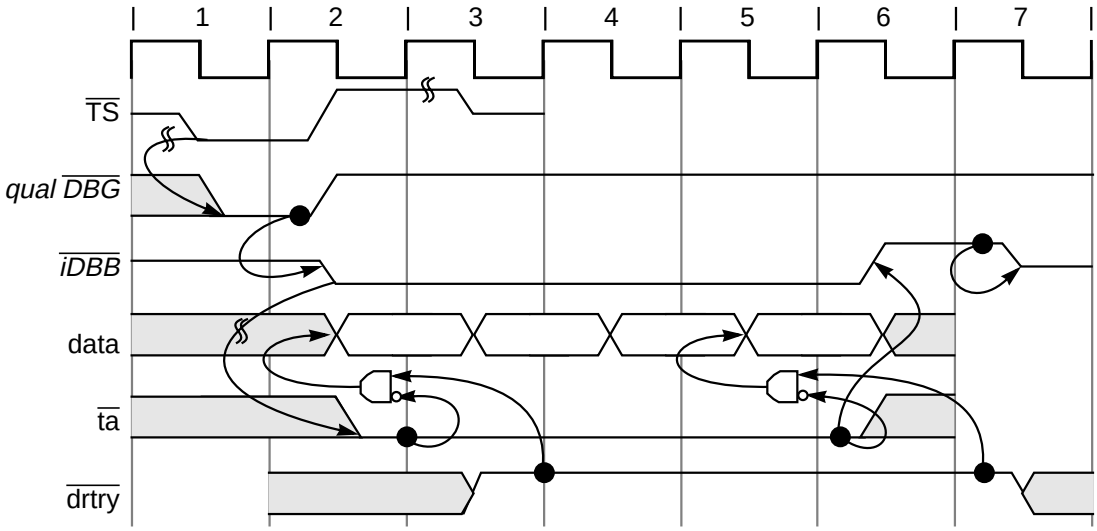


Figure 8-12. Normal Burst Transaction

For read bursts, $\overline{DRTRY}$ may be asserted one bus clock cycle after $\overline{TA}$ is asserted to signal that the data presented with $\overline{TA}$ is invalid and that the processor must wait for the negation of $\overline{DRTRY}$ before forwarding data to the processor (see Figure 8-13). Thus, a data beat can be terminated by a predicted branch with $\overline{TA}$ and then one bus clock cycle later confirmed with the negation of $\overline{DRTRY}$. The $\overline{DRTRY}$ signal is valid only for read transactions. $\overline{TA}$ must be asserted on the bus clock cycle before the first bus clock cycle of the assertion of $\overline{DRTRY}$; otherwise the results are undefined.

The $\overline{DRTRY}$ signal extends data bus mastership such that other processors cannot use the data bus until $\overline{DRTRY}$ is negated. Therefore, in the example inFigure 8-13, data bus tenure for the next transaction cannot begin until bus clock cycle 6. This is true for both read and write operations even though $\overline{DRTRY}$ does not extend bus mastership for write operations.

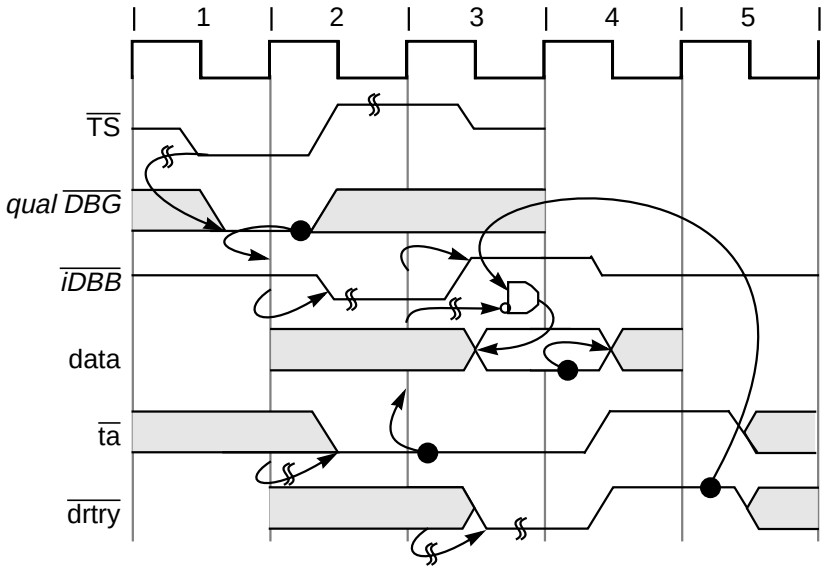


Figure 8-13. Termination with DRTRY

Figure 8-14 shows the effect of using DRTRY during a burst read. It also shows the effect of using TA to pace the data transfer rate. Notice that in bus clock cycle 3 of Figure 8-15 on Page 8-25, TA is negated for the second data beat. Gekko data pipeline does not proceed until bus clock cycle 4 when the TA is reasserted.

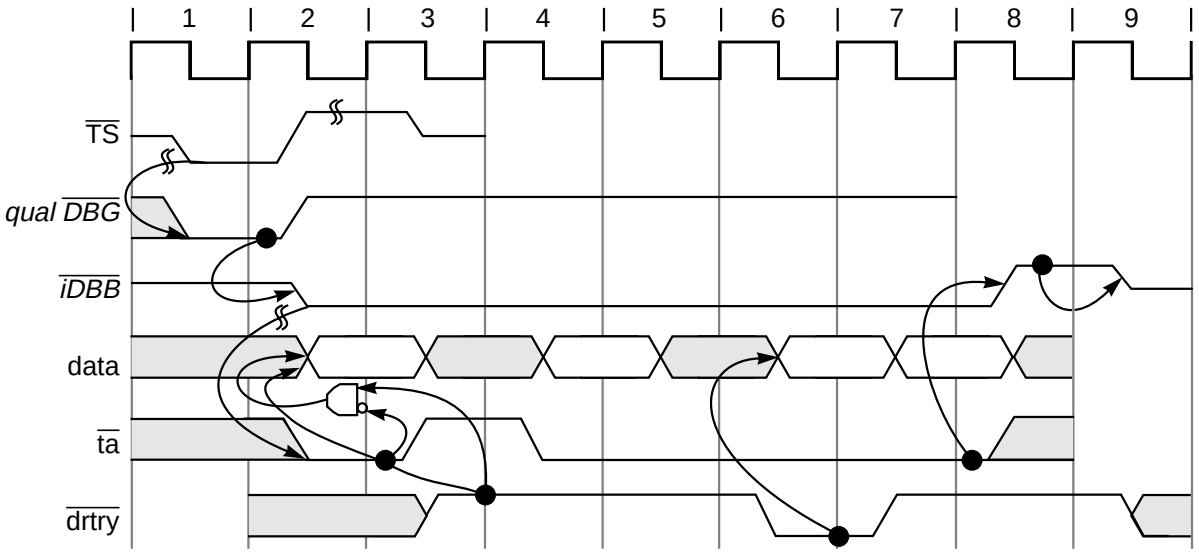


Figure 8-14. Read Burst with TA Wait States and DRTRY

NOTE: DRTRY is useful for systems that implement predicted forwarding of data such as those with direct-mapped, third-level caches where hit/miss is determined on the following bus clock cycle, or for parity- or ECC-checked memory systems. Also note that DRTRY may not be implemented on other PowerPC processors.

8.4.3.2 Data Transfer Termination Due to a Bus Error

The $\overline{\text{TEA}}$ signal indicates that a bus error occurred. It may be asserted during data bus tenure. Asserting $\overline{\text{TEA}}$ to Gekko terminates the transaction; that is, further assertions of $\overline{\text{TA}}$ are ignored and the data bus tenure is terminated.

Assertion of the $\overline{\text{TEA}}$ signal causes a machine check exception (and possibly a checkstop condition within Gekko). The hard reset exception is a nonrecoverable, nonmaskable asynchronous exception. When HRESET is asserted or at power-on reset (POR), the 750 immediately branches to 0xFFF0_0100 without attempting to reach a recoverable state. A hard reset has the highest priority of any exception. It is always nonrecoverable.

Table 4-9 on Page 4-16 shows the state of the machine just before it fetches the first instruction of the system reset handler after a hard reset. In Table 4-9, the term “Unknown” means that the content may have been disordered. These facilities must be properly initialized before use. The FPRs, BATs, and TLBs may have been disordered. To initialize the BATs, first set them all to zero, then to the correct values before any address translation occurs..” Note also that Gekko does not implement a synchronous error capability for memory accesses. This means that the exception instruction pointer saved into the SRR0 register does not point to the memory operation that caused the assertion of $\overline{\text{TEA}}$, but to the instruction about to be executed (perhaps several instructions later). However, assertion of $\overline{\text{TEA}}$ does not invalidate data entering the GPR or the cache. Additionally, the address corresponding to the access that caused $\overline{\text{TEA}}$ to be asserted is not latched by Gekko. To recover, the exception handler must determine and remedy the cause of the $\overline{\text{TEA}}$, or Gekko must be reset; therefore, this function should only be used to indicate fatal system conditions to the processor.

After Gekko has committed to run a transaction, that transaction must eventually complete. Address retry causes the transaction to be restarted; $\overline{\text{TA}}$ wait states and $\overline{\text{DRTRY}}$ assertion for reads delay termination of individual data beats. Eventually, however, the system must either terminate the transaction or assert the $\overline{\text{TEA}}$ signal. For this reason, care must be taken to check for the end of physical memory and the location of certain system facilities to avoid memory accesses that result in the assertion of $\overline{\text{TEA}}$.

Note that $\overline{\text{TEA}}$ generates a machine check exception depending on MSR[ME]. Clearing the machine check exception enable control bits leads to a true checkstop condition (instruction execution halted and processor clock stopped).

8.4.4 Memory Coherency—MEI Protocol

Gekko provides dedicated hardware to provide memory coherency by snooping bus transactions. The address retry capability enforces the three-state, MEI cache-coherency protocol (see Figure 8-15 on Page 8-25).

The global ($\overline{\text{GBL}}$) output signal indicates whether the current transaction must be snooped by other snooping devices on the bus. Address bus masters assert $\overline{\text{GBL}}$ to indicate that the current transaction is a global access (that is, an access to memory shared by more than one device). If $\overline{\text{GBL}}$ is not asserted for the transaction, that transaction is not snooped. When other devices detect the $\overline{\text{GBL}}$ input asserted, they must respond by snooping the broadcast address.

Normally, $\overline{\text{GBL}}$ reflects the M bit value specified for the memory reference in the corresponding translation descriptor(s). Note that care must be taken to minimize the number of pages marked as global, because the retry protocol discussed in the previous section is used to enforce coherency and can require significant bus bandwidth.

When Gekko is not the address bus master, $\overline{\text{GBL}}$ is an input. Gekko snoops a transaction if $\overline{\text{TS}}$ and $\overline{\text{GBL}}$ are asserted together in the same bus clock cycle (this is a qualified snooping condition). No snoop update to Gekko cache occurs if the snooped transaction is not marked global. This includes invalidation cycles.

When Gekko detects a qualified snoop condition, the address associated with the $\overline{TS}$ is compared against the data cache tags. Snooping completes if no hit is detected. If, however, the address hits in the cache, Gekko reacts according to the MEI protocol shown in Figure 8-15, assuming the WIM bits are set to write-back, caching-allowed, and coherency-enforced modes (WIM = 001).

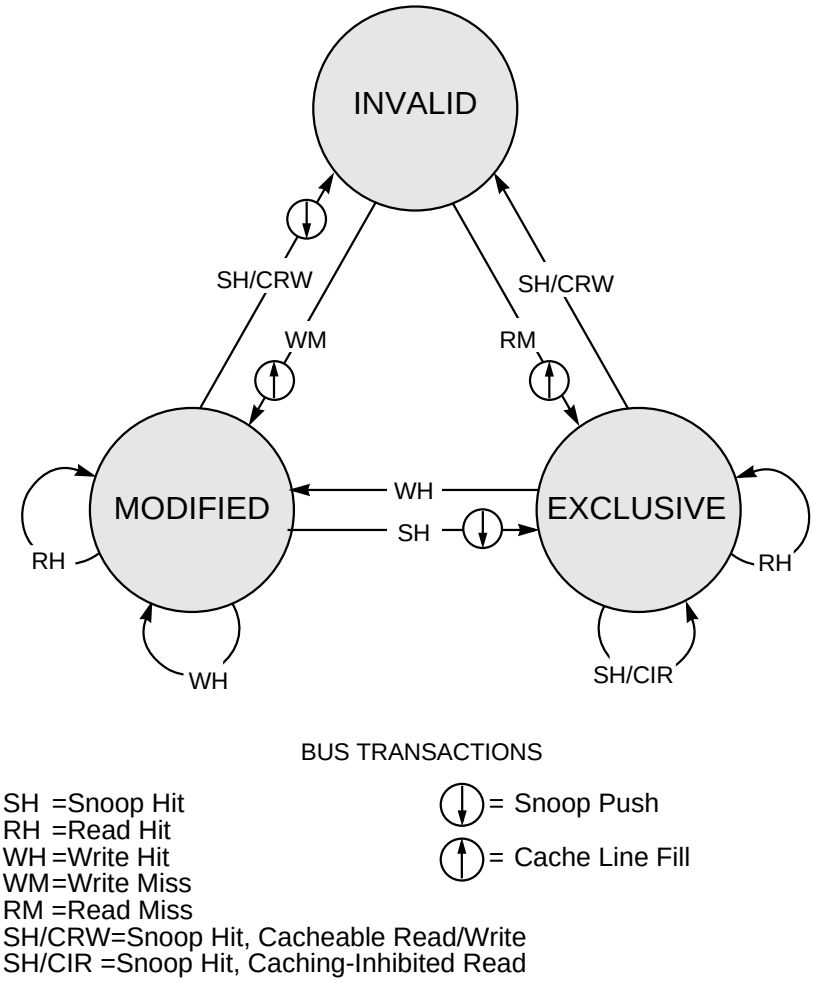


Figure 8-15. MEI Cache Coherency Protocol—State Diagram (WIM = 001)

8.5 Timing Examples

This section shows timing diagrams for various scenarios. Figure 8-16 on Page 8-26 illustrates the fastest single-beat reads possible for Gekko. This figure shows both minimal latency and maximum single-beat throughput. By delaying the data bus tenure, the latency increases, but, because of split-transaction pipelining, the overall throughput is not affected unless the data bus latency causes the third address tenure to be delayed.

Note that all bidirectional signals are three-stated between bus tenures.

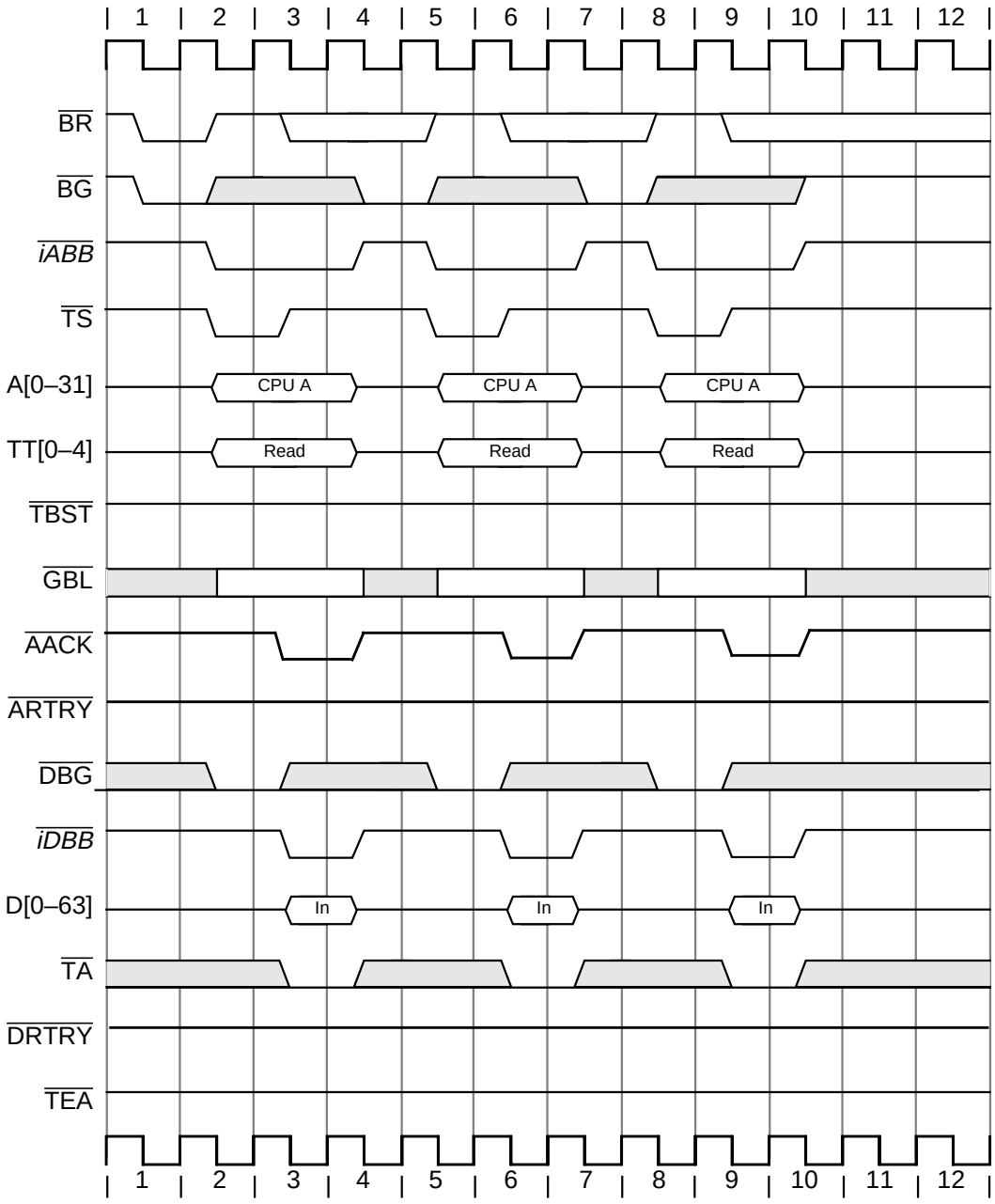


Figure 8-16. Fastest Single-Beat Reads

Figure 8-17 illustrates the fastest single-beat writes supported by Gekko. All bidirectional signals are three-stated between bus tenures.

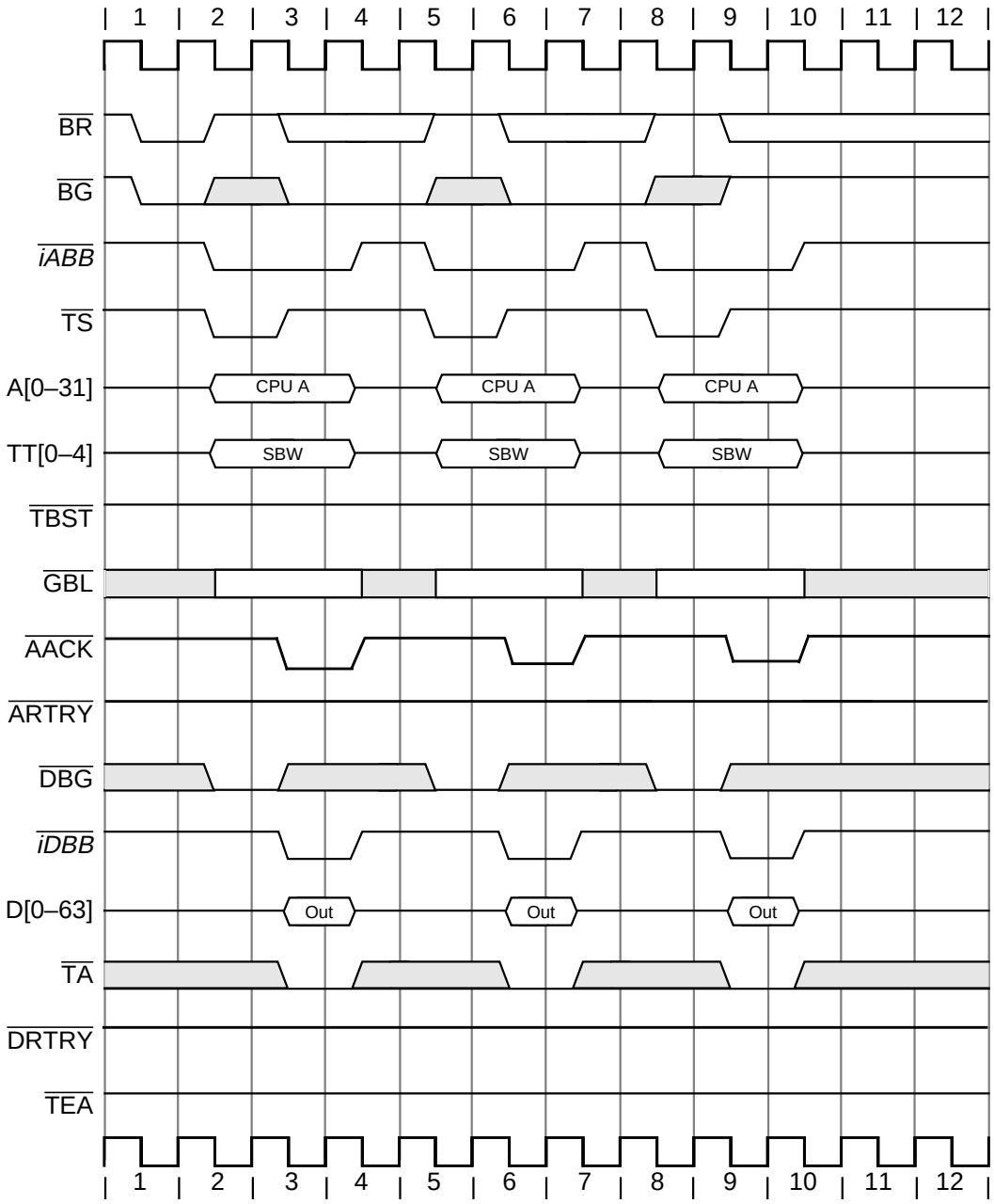


Figure 8-17. Fastest Single-Beat Writes

Figure 8-18 on Page 8-28 shows three ways to delay single-beat reads showing data-delay controls:

- The $\overline{TA}$ signal can remain negated to insert wait states in clock cycles 3 and 4.
- For the second access, $\overline{DBG}$ could have been asserted in clock cycle 6.
- In the third access, $\overline{DRTRY}$ is asserted in clock cycle 11 to flush the previous data.

NOTE: All bidirectional signals are three-stated between bus tenures. The pipelining shown in Figure 8-18 can occur if the second access is not another load (for example, an instruction fetch).

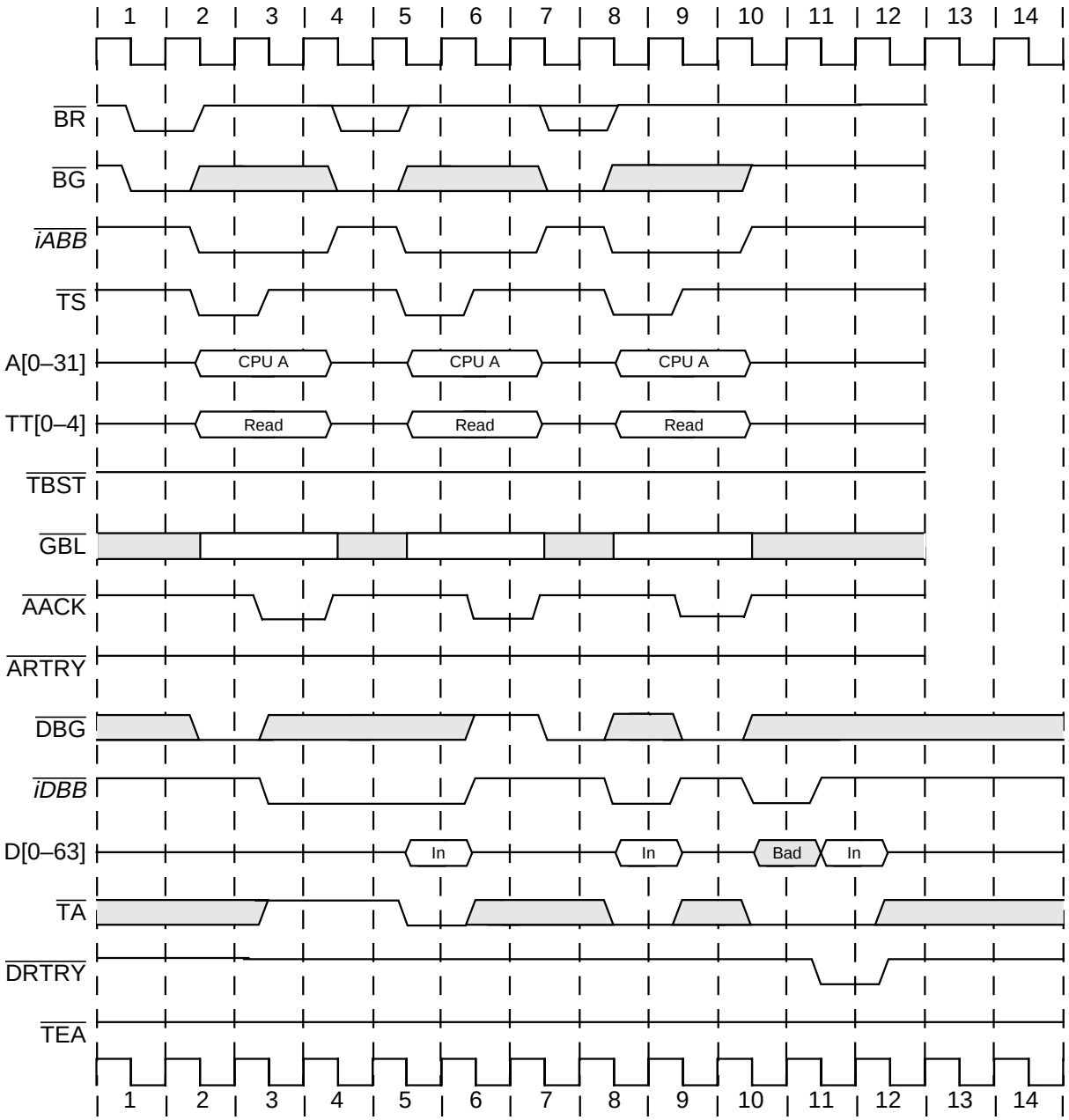


Figure 8-18. Single-Beat Reads Showing Data-Delay Controls

Figure 8-19 on Page 8-29 shows data-delay controls in a single-beat write operation. Note that all bidirectional signals are three-stated between bus tenures. Data transfers are delayed in the following ways:

- The $\overline{TA}$ signal is held negated to insert wait states in clocks 3 and 4.

- In clock 6, $\overline{\text{DBG}}$ is held negated, delaying the start of the data tenure.

The last access is not delayed ($\overline{\text{DRTRY}}$ is valid only for read operations).

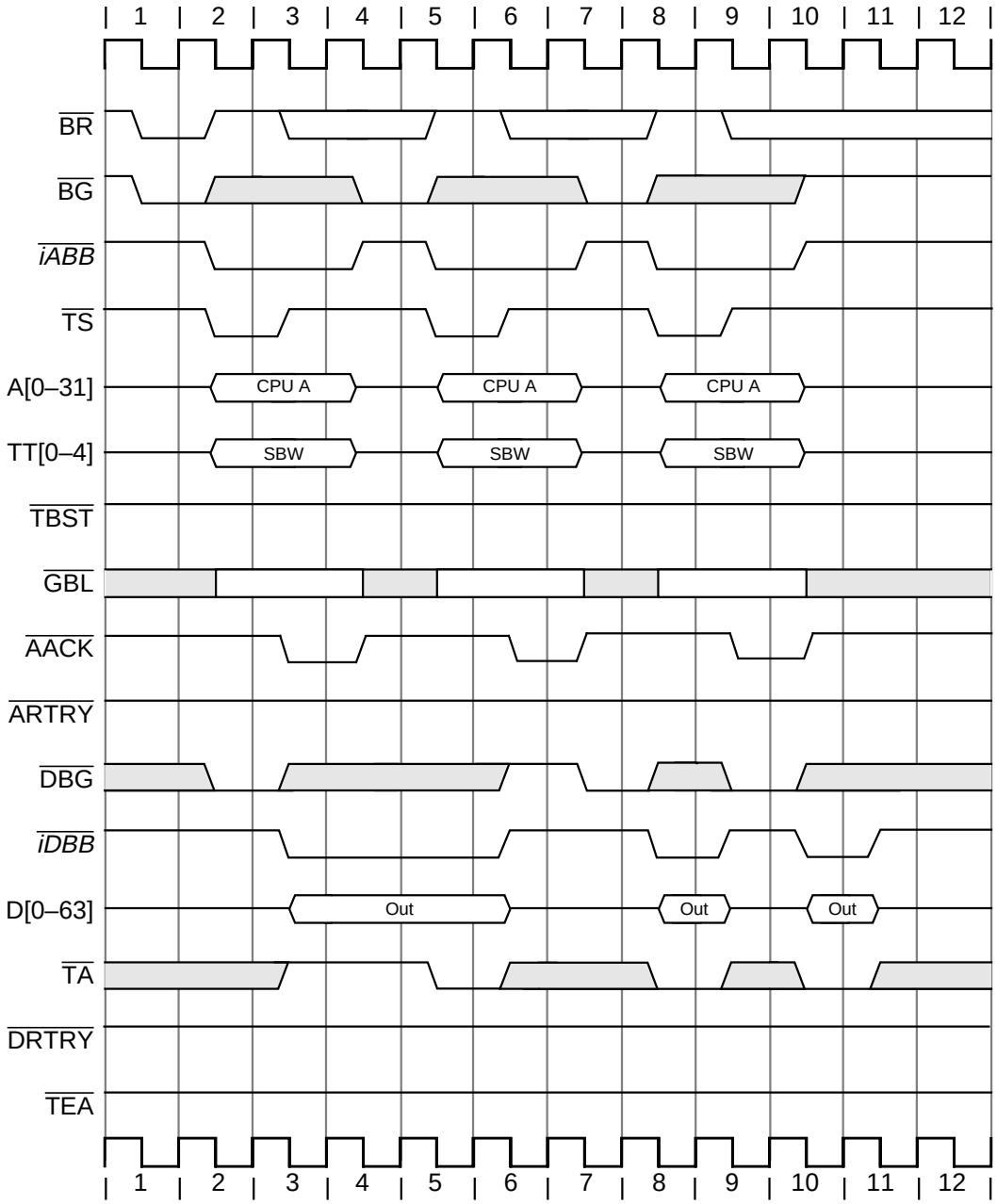


Figure 8-19. Single-Beat Writes Showing Data Delay Controls

Figure 8-20 on Page 8-30 shows the use of data-delay controls with burst transfers. Note that all bidirectional signals are three-stated between bus tenures. Note the following:

- The first data beat of bursted read data (clock 0) is the critical quad word.
- The write burst shows the use of $\overline{\text{TA}}$ signal negation to delay the third data beat.
- The final read burst shows the use of $\overline{\text{DRTRY}}$ on the third data beat.

- The address for the third transfer is delayed until the first transfer completes.

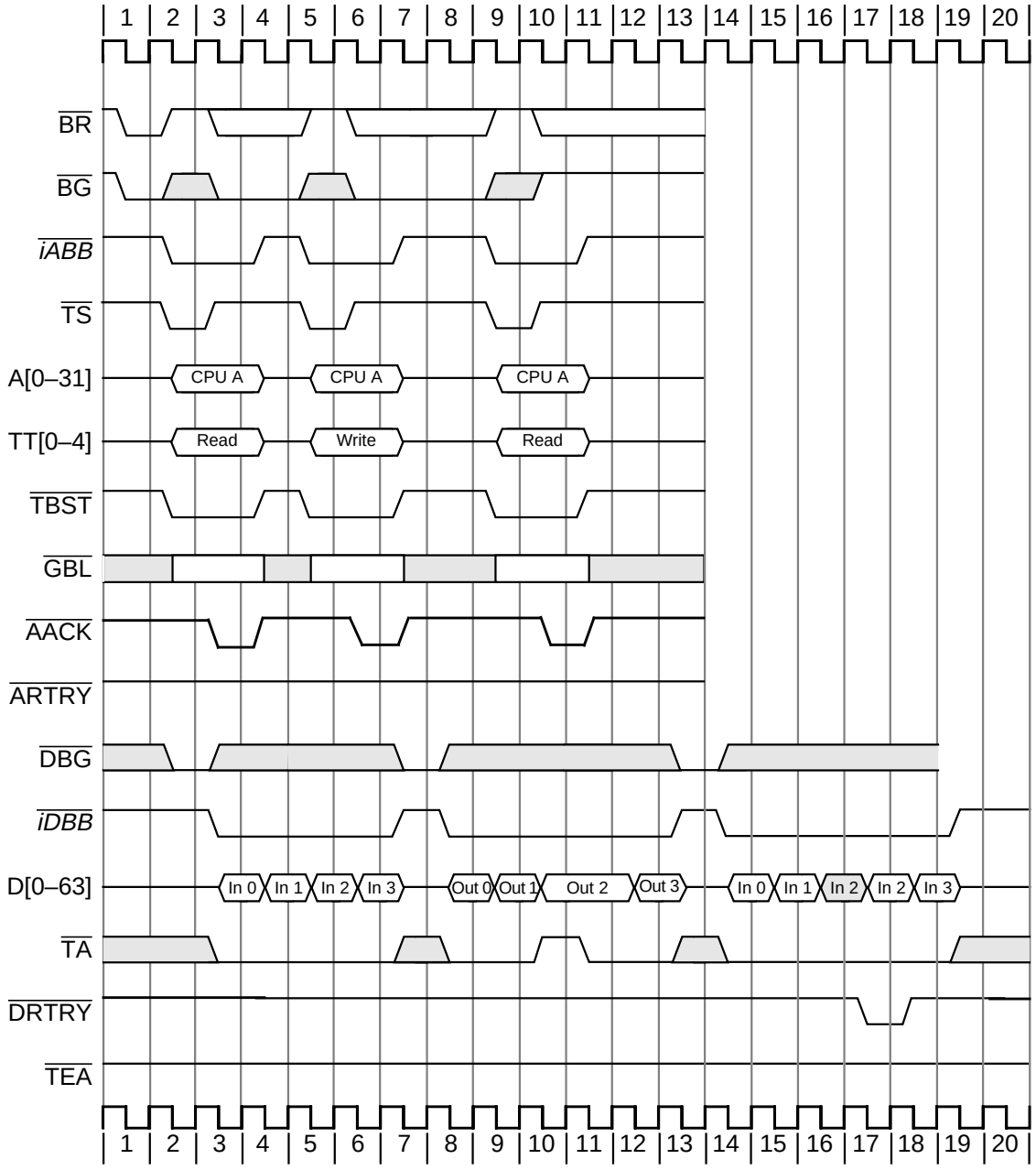


Figure 8-20. Burst Transfers with Data Delay Controls

Figure 8-21 on Page 8-31 shows the use of the $\overline{\text{TEA}}$ signal. Note that all bidirectional signals are three-stated between bus tenures. Note the following:

- The first data beat of the read burst (in clock 0) is the critical quad word.
- The $\overline{\text{TEA}}$ signal truncates the burst write transfer on the third data beat.
- Gekko eventually causes an exception to be taken on the $\overline{\text{TEA}}$ event.

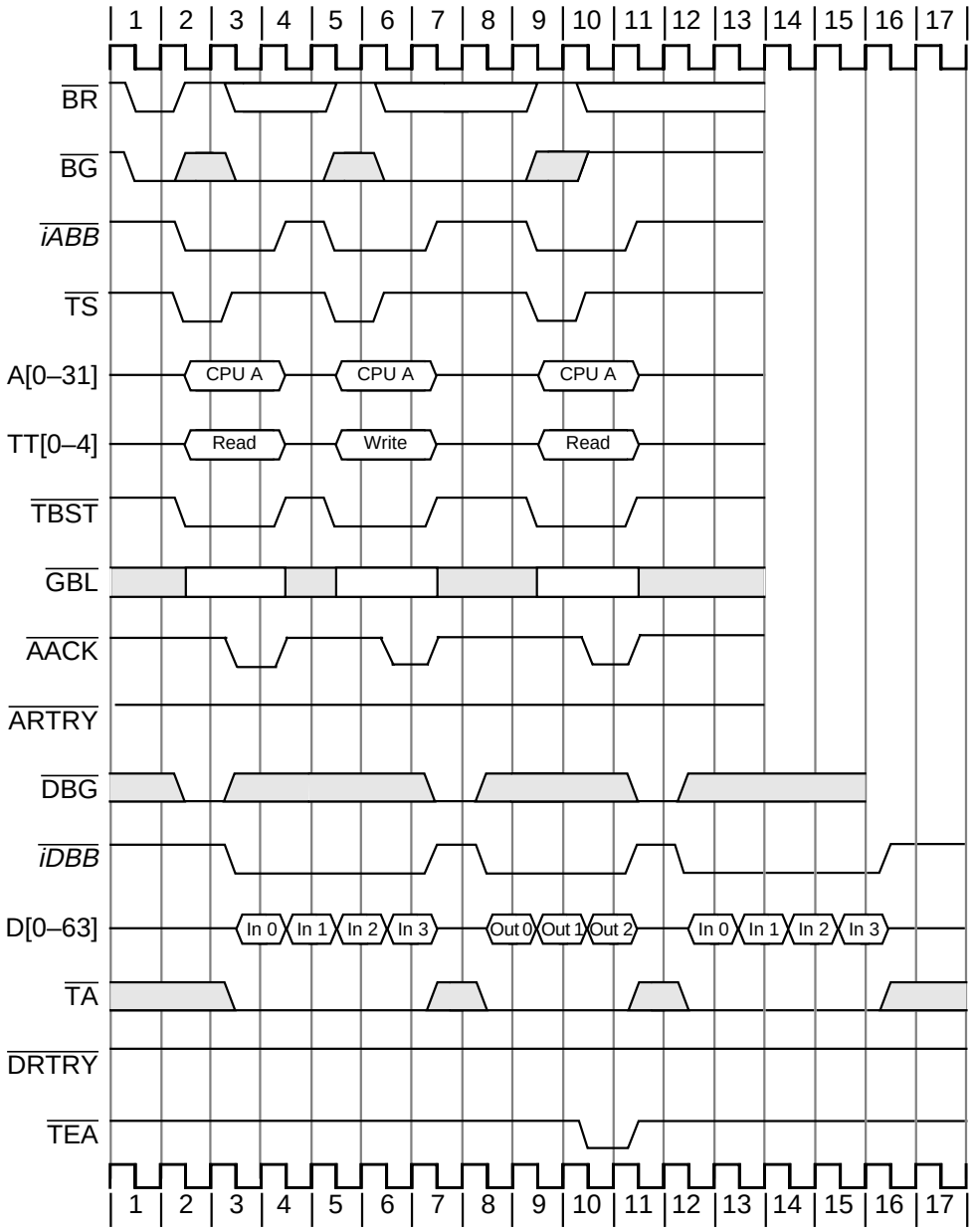


Figure 8-21. Use of Transfer Error Acknowledge ($\overline{\text{TEA}}$)

8.6 No-$\overline{\text{DRTRY}}$ Bus Configuration

Gekko is internally configured for the no-$\overline{\text{DRTRY}}$ mode. The no-$\overline{\text{DRTRY}}$ mode allows the forwarding of data during load operations to the internal CPU one bus cycle sooner than in the normal bus protocol.

The 60x bus protocol specifies that, during load operations, the memory system normally has the capability to cancel data that was read by the master on the bus cycle after $\overline{\text{TA}}$ was asserted. This late cancellation protocol requires Gekko to hold any loaded data at the bus interface for one additional bus clock to verify that the data is valid before forwarding it to the internal CPU. Gekko uses the no-$\overline{\text{DRTRY}}$ mode that eliminates this one-cycle stall during all load operations, and allows for the forwarding of data to the internal CPU immediately when $\overline{\text{TA}}$ is recognized.

The data can no longer be cancelled the cycle after it is acknowledged by an assertion of $\overline{TA}$. Data is immediately forwarded to the CPU internally, and any attempt at late cancellation by the system may cause improper operation by Gekko.

When a typical 60x PowerPC processor is following normal bus protocol, data may be cancelled the bus cycle after $\overline{TA}$ by either of two means—late cancellation by $\overline{DRTRY}$, or late cancellation by $\overline{ARTRY}$. For Gekko, due to the default no-$\overline{DRTRY}$ mode, both late cancellation cases must be disallowed in the system design for the bus protocol. The system must also ensure that an assertion of $\overline{ARTRY}$ by a snooping device must occur before or coincident with the first assertion of $\overline{TA}$ to Gekko, but not on the cycle after the first assertion of $\overline{TA}$.

8.7 32-bit Data Bus Mode

Gekko supports an optional 32-bit data bus mode. The 32-bit data bus mode operates the same as the 64-bit data bus mode with the exception of the byte lanes involved in the transfer and the number of data beats that are performed. When in 32-bit data bus mode, only byte lanes 0 through 3 are used corresponding to DH0–DH31 and DP0–DP3. Byte lanes 4 through 7 corresponding to DL0–DL31 and DP4–DP7 are never used in this mode. The unused data bus signals are not sampled by Gekko during read operations, and they are driven low during write operations.

The number of data beats required for a data tenure in the 32-bit data bus mode is one, two, or eight beats depending on the size of the program transaction and the cache mode for the address. Data transactions of one or two data beats are performed for caching-inhibited load/store or write-through store operations. These transactions do not assert the $\overline{TBST}$ signal even though a two-beat burst may be performed (having the same $\overline{TBST}$ and
TSIZ[0–2] encodings as the 64-bit data bus mode). Single-beat data transactions are performed for bus operations of 4 bytes or less, and double-beat data transactions are performed for 8-byte operations only. Gekko only generates an 8-byte operation for a double-word-aligned load or store double operation to or from the floating-point registers. All cache-inhibited instruction fetches are performed as word (single-beat) operations.

Data transactions of eight data beats are performed for burst operations that load into or store from Gekko's internal caches. These transactions transfer 32 bytes in the same way as in 64-bit data bus mode, asserting the $\overline{TBST}$ signal, and signaling a transfer size of 2 (TSIZ(0–2) = 0b010).

The same bus protocols apply for arbitration, transfer, and termination of the address and data tenures in the 32-bit data bus mode as they apply to the 64-bit data bus mode. Late $\overline{ARTRY}$ cancellation of the data tenure applies on the bus clock after the first data beat is acknowledged (after the first $\overline{TA}$) for word or smaller transactions, or on the bus clock after the second data beat is acknowledged (after the second $\overline{TA}$) for double-word or burst operations (or coincident with respective $\overline{TA}$ if no-$\overline{DRTRY}$ mode is selected).

An example of an eight-beat data transfer while Gekko is in 32-bit data bus mode is shown in Figure 8-22 on Page 8-33.

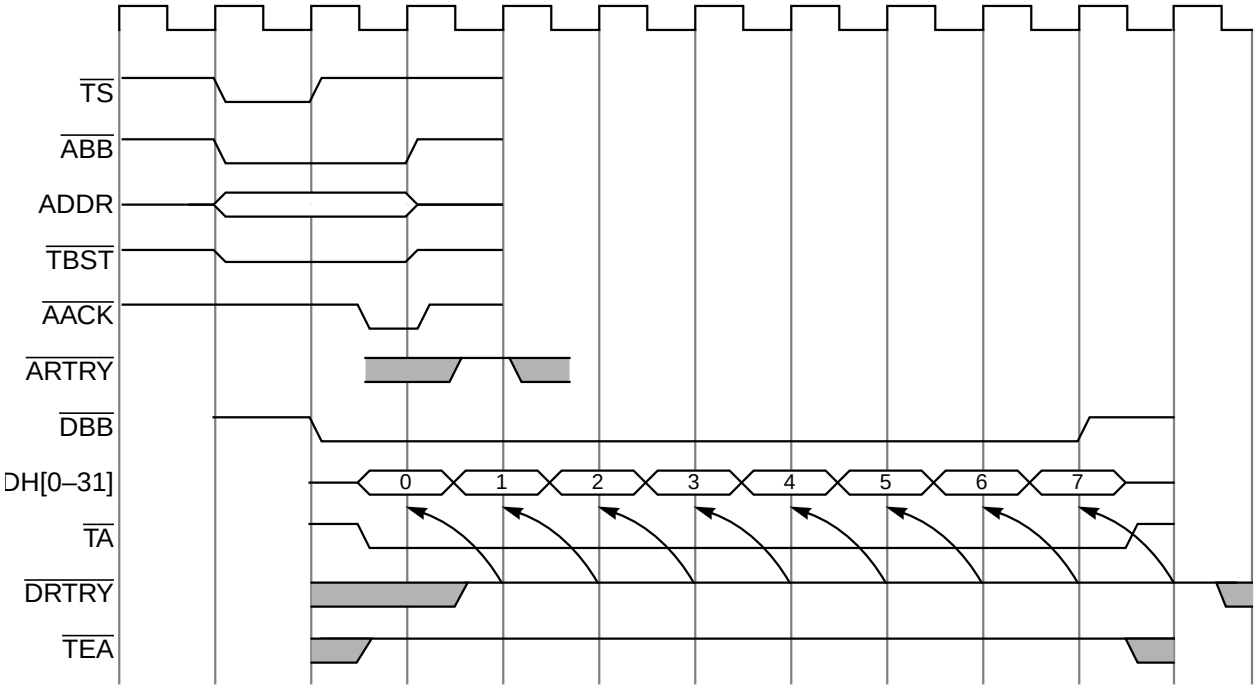


Figure 8-22. 32-Bit Data Bus Transfer (Eight-Beat Burst)

An example of a two-beat data transfer is shown in Figure 8-23.

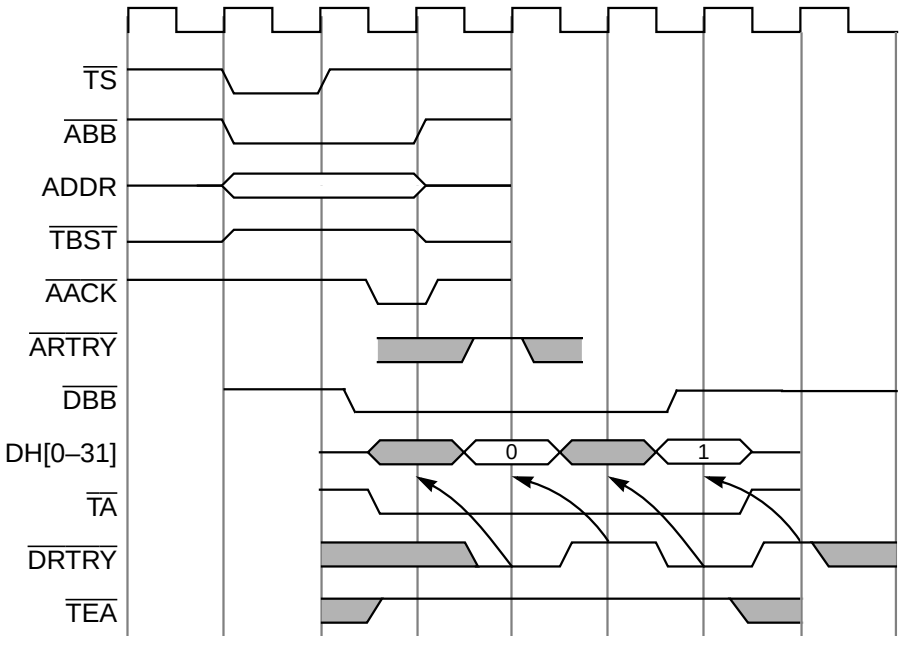


Figure 8-23. 32-Bit Data Bus Transfer (Two-Beat Burst with $\overline{\text{DRTRY}}$)

Gekko selects 64-bit or 32-bit data bus mode at startup by sampling the state of the $\overline{\text{QACK}}$ signal at the negation of $\overline{\text{HRESET}}$. If the $\overline{\text{QACK}}$ signal is asserted at the negation of $\overline{\text{HRESET}}$, 64-bit data

mode is selected by Gekko. If $\overline{\text{QACK}}$ is de-asserted at the negation of $\overline{\text{HRESET}}$, 32-bit data mode is selected. Table 8-5 describes the burst ordering when Gekko is in 32-bit mode.

Table 8-5. Burst Ordering—32-Bit Bus

Data Transfer	For Starting Address:			
	A[27–28] = 00	A[27–28] = 01	A[27–28] = 10	A[27–28] = 11
First data beat	DW0-U	DW1-U	DW2-U	DW3-U
Second data beat	DW0-L	DW1-L	DW2-L	DW3-L
Third data beat	DW1-U	DW2-U	DW3-U	DW0-U
Fourth data beat	DW1-L	DW2-L	DW3-L	DW0-L
Fifth data beat	DW2-U	DW3-U	DW0-U	DW1-U
Sixth data beat	DW2-L	DW3-L	DW0-L	DW1-L
Seventh data beat	DW3-U	DW0-U	DW1-U	DW2-U
Eighth data beat	DW3-L	DW0-L	DW1-L	DW2-L

Notes: A[29–31] are always 0b000 for burst transfers by the 750.

"U" and "L" represent the upper and lower word of the double word respectively.

The aligned data transfer cases for 32-bit data bus mode are shown in Table 8-6. All of the transfers require a single data beat (if caching-inhibited or write-through) except for double-word cases which require two data beats. The double-word case is only generated by Gekko for load or store double operations to/from the floating-point registers. All caching-inhibited instruction fetches are performed as word operations.

Table 8-6. Aligned Data Transfers (32-Bit Bus Mode)

Transfer Size	TSIZ0	TSIZ1	TSIZ2	A[29–31]	Data Bus Byte Lane(s)							
					0	1	2	3	4	5	6	7
Byte	0	0	1	000	A	—	—	—	x	x	x	x
	0	0	1	001	—	A	x	—	x	x	x	x
	0	0	1	010	—	—	A	—	x	x	x	x
	0	0	1	011	—	—	—	A	x	x	x	x
	0	0	1	100	A	—	—	—	x	x	x	x
	0	0	1	101	—	A	—	—	x	x	x	x
	0	0	1	110	—	—	A	—	x	x	x	x
	0	0	1	111	—	—	—	A	x	x	x	x

Table 8-6. Aligned Data Transfers (32-Bit Bus Mode)

Transfer Size	TSIZ0	TSIZ1	TSIZ2	A[29–31]	Data Bus Byte Lane(s)							
					0	1	2	3	4	5	6	7
Half word	0	1	0	000	A	A	—	—	x	x	x	x
	0	1	0	010	—	—	A	A	x	x	x	x
	0	1	0	100	A	A	—	—	x	x	x	x
	0	1	0	110	—	—	A	A	x	x	x	x
Word	1	0	0	000	A	A	A	A	x	x	x	x
	1	0	0	100	A	A	A	A	x	x	x	x
Double word	0	0	0	000	A	A	A	A	x	x	x	x
Second beat	0	0	0	000	A	A	A	A	x	x	x	x

Notes:

A: Byte lane used
—: Byte lane not used
x: Byte lane not used in 32-bit bus mode

Misaligned data transfers in the 32-bit bus mode is the same as in the 64-bit bus mode with the exception that only DH[0-31] data lines are used.Table 8-7 on Page 8-36 shows examples of 4-byte mis-aligned transfers starting at each possible byte address within a double word.

Table 8-7. Misaligned 32-Bit Data Bus Transfer (Four-Byte Examples)

Transfer Size (Four Bytes)	TSIZ[0–2]	A[29–31]	Data Bus Byte Lanes							
			0	1	2	3	4	5	6	7
Aligned	1 0 0	0 0 0	A	A	A	A	x	x	x	x
Misaligned—first access	0 1 1	0 0 1		A	A	A	x	x	x	x
second access	0 0 1	1 0 0	A	—	—	—	x	x	x	x
Misaligned—first access	0 1 0	0 1 0	—	—	A	A	x	x	x	x
second access	0 1 0	1 0 0	A	A	—	x	x	x	x	x
Misaligned—first access	0 0 1	0 1 1	—	—	—	A	x	x	x	x
second access	0 1 1	1 0 0	A	A	A	—	x	x	x	x
Aligned	1 0 0	1 0 0	A	A	A	A	x	x	x	x
Misaligned—first access	0 1 1	1 0 1	—	A	A	A	x	x	x	x
second access	0 0 1	0 0 0	A	—	—	—	x	x	x	x
Misaligned—first access	0 1 0	1 1 0	—	—	A	A	x	x	x	x
second access	0 1 0	0 0 0	A	A	—	—	x	x	x	x
Misaligned—first access	0 0 1	1 1 1	—	—	—	A	x	x	x	x
second access	0 1 1	0 0 0	A	A	A	—	x	x	x	x

Notes:

A: Byte lane used
—: Byte lane not used
x: Byte lane not used in 32-bit bus mode

8.8 Interrupt, Checkstop, and Reset Signals

This section describes external interrupts, checkstop operations, and hard and soft reset inputs.

8.8.1 External Interrupts

The external interrupt input signals ($\overline{\text{INT}}$ and $\overline{\text{MCP}}$) of Gekko eventually force the processor to take the external interrupt vector if the MSR[EE] is set, or the machine check interrupt if the MSR[ME] and the HID0[EMCP] bits are set.

8.8.2 Checkstops

A checkstop causes the processor to halt. Once Gekko enters a checkstop state, only a hard reset can clear the processor from the checkstop state.

Gekko has two checkstop input signals—$\overline{\text{CKSTP\_IN}}$ (nonmaskable) and $\overline{\text{MCP}}$ (enabled when MSR[ME] is cleared, and HID0[EMCP] is set). If $\overline{\text{CKSTP\_IN}}$ or $\overline{\text{MCP}}$ is asserted, Gekko halts operations by gating off all internal clocks.

Following is the list of checkstop sources:

- Machine Check with MSR(ME)=0. If MSR(ME)=0 when a machine check interrupt occurs, then the checkstop state is entered. The machine check sources are as follows.
  - — TEA_ assertion on the 60X bus
  - — Address parity error on the 60X bus
  - — Data parity error on the 60X bus
  - — Data double bit error on the L2 bus
- Machine check input pin (MCP_)
- Checkstop input pin (CKSTP_IN_)

8.8.3 Reset Inputs

Gekko has two reset inputs, described as follows:

- $\overline{\text{HRESET}}$ (hard reset)—The $\overline{\text{HRESET}}$ signal is used for power-on reset sequences, or for situations in which Gekko must go through the entire cold start sequence of internal hardware initializations.
- $\overline{\text{SRESET}}$ (soft reset)—The soft reset input provides warm reset capability. This input can be used to avoid forcing Gekko to complete the cold start sequence.

When either $\overline{\text{HRESET}}$ is negated or $\overline{\text{SRESET}}$ transitions to asserted, the processor attempts to fetch code from the system reset exception vector. The vector is located at offset 0x00100 from the exception prefix (all zeros or ones, depending on the setting of the exception prefix bit in the machine state register (MSR[IP]). The MSR[IP] bit is set for $\overline{\text{HRESET}}$.

8.8.4 System Quiesce Control Signals

The system quiesce control signals ($\overline{\text{QREQ}}$ and $\overline{\text{QACK}}$) allow the processor to enter the nap or sleep low-power states, and bring bus activity to a quiescent state in an orderly fashion.

Prior to entering the nap or sleep power state, Gekko asserts the $\overline{\text{QREQ}}$ signal. This signal allows the system to terminate or pause any bus activities that are normally snooped. When the system is ready to enter the system quiesce state, it asserts the $\overline{\text{QACK}}$ signal. At this time Gekko may enter a quiescent (low power) state. When Gekko is in the quiescent state, it stops snooping bus activity. While Gekko is in the nap power state, the system power controller can enable snooping by Gekko by deasserting the $\overline{\text{QACK}}$ signal for at least eight bus clock cycles, after which Gekko is capable of snooping bus transactions. The reassertion of $\overline{\text{QACK}}$ following the snoop transactions will cause Gekko to reenter the nap power state.

8.9 Processor State Signals

This section describes Gekko's support for atomic update and memory through the use of the **lwarx**/**stwcx.** opcode pair, and includes a description of the $\overline{\text{TLBISYNC}}$ input.

8.9.1 Support for the lwarx/stwcx. Instruction Pair

The Load Word and Reserve Indexed (**lwarx**) and the Store Word Conditional Indexed (**stwcx.**) instructions provide a means for atomic memory updating. Memory can be updated atomically by setting a reservation on the load and checking that the reservation is still valid before the store is performed. In Gekko, the reservations are made on behalf of aligned, 32-byte sections of the memory address space.

8.9.2 $\overline{\text{TLBISYNC}}$ Input

Note: The $\overline{\text{TLBISYNC}}$ pin is not connected in Gekko, so the corresponding MMU synchronization function is effectively disabled.

8.10 IEEE 1149.1a-1993 Compliant Interface

Gekko boundary-scan interface is a fully-compliant implementation of the IEEE 1149.1a-1993 standard. This section describes Gekko's IEEE 1149.1a-1993 (JTAG) interface.

8.10.1 JTAG/COP Interface

Gekko has extensive on-chip test capability including the following:

- Debug control/observation (COP)
- Boundary scan (standard IEEE 1149.1a-1993 (JTAG) compliant interface)
- Support for manufacturing test

The COP and boundary scan logic are not used under typical operating conditions. Detailed discussion of Gekko test functions is beyond the scope of this document; however, sufficient information has been provided to allow the system designer to disable the test functions that would impede normal operation.

The JTAG/COP interface is shown in Figure 8-24. For more information, refer to *IEEE Standard Test Access Port and Boundary Scan Architecture IEEE STD 1149-1a-1993*.

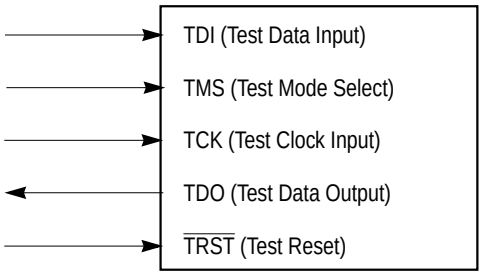


Figure 8-24. IEEE 1149.1a-1993 Compliant Boundary Scan Interface

Chapter 9 L2 Cache, Locked D-Cache, DMA and Write Gather Pipe

This chapter describes Gekko microprocessor‘s implementation of L2 cache, L1 D-cache partition, direct memory access (DMA) and write gather pipe.

9.1 L2 Cache

Gekko’s L2 cache is implemented with an on-chip, two-way set-associative tag memory with 2048 tags per way, and an on-chip 256 Kbyte SRAM for data storage. The tags are sectored to support two cache blocks per tag entry (two sectors, 64 bytes). Each sector (32-byte L1 cache block) in the L2 cache has its own valid and modified bits. In addition, the SRAM includes an 8-bit ECC for every double word. The ECC logic corrects single bit errors and detects double bit errors as data is read from the SRAM. The L2 cache maintains cache coherency through snooping and is normally configured to operate in copy-back mode.

The L2 cache control register (L2CR) allows control of:

- L2 cache configuration
- Double bit error machine check
- Global invalidation of L2 contents
- Write-through operation
- L2 test support

9.1.1 L2 Cache Operation

Gekko’s L2 cache is a combined instruction and data cache that receives memory requests from both L1 instruction and data caches independently. The L1 requests are generally the result of instruction fetch misses, data load or store misses, write-through operations, or cache management instructions. Each L1 request generates an address lookup in the L2 tags. If a hit occurs, the instructions or data are forwarded to the L1 cache. A miss in the L2 tags causes the L1 request to be forwarded to the 60x bus interface. The cache block received from the bus is forwarded to the L1 cache immediately, and is also loaded into the L2 cache with the tag marked valid and unmodified. If the cache block loaded into the L2 causes a new tag entry to be allocated and the current tag entry is marked valid modified, the modified sectors of the tag to be replaced are castout from the L2 cache to the 60x bus.

At any given time the L1 instruction cache may have one instruction fetch request, and the L1 data cache may have one load and two stores requesting L2 cache access. The L2 cache also services snoop requests from the 60x bus. When there are multiple pending requests to the L2 cache, snoop requests have highest priority, followed by data load and store requests (serviced on a first-in, first-out basis). Instruction fetch requests have the lowest priority in accessing the L2 cache when there are multiple accesses pending.

If read requests from both the L1 instruction and data caches are pending, the L2 cache can perform hit-under-miss and supplies the available instruction or data while a bus transaction for the previous L2 cache miss is performed. The L2 cache does not support miss-under-miss, and the second instruction fetch or data load stalls until the bus operation resulting from the first L2 miss completes.

All requests to the L2 cache that are marked cacheable (even if the respective L1 cache is disabled or locked) cause tag lookup and will be serviced if the instructions or data are in the L2 cache. Burst and single-beat read requests from the L1 caches that hit in the L2 cache are forwarded instructions

or data, and the L2 LRU bit for that tag is updated. Burst writes from the L1 data cache due to a castout or replacement copyback are written only to the L2 cache, and the L2 cache sector is marked modified.

If the L2 cache is configured as write-through, the L2 sector is marked unmodified, and the write is forwarded to the 60x bus. If the L1 castout requires a new L2 tag entry to be allocated and the current tag is marked modified, any modified sectors of the tag to be replaced are cast out of the L2 cache to the 60x bus.

Single-beat read requests from the L1 caches that miss in the L2 cache do not cause any state changes in the L2 cache and are forwarded on the 60x bus interface. Cacheable single-beat store requests marked copy-back that hit in the L2 are allowed to update the L2 cache sector, but do not cause L2 cache sector allocation or deallocation. Cacheable, single-beat store requests that miss in the L2 are forwarded to the 60x bus. Single-beat store requests marked write-through (through address translation or through the configuration of L2CR[L2WT]) are written to the L2 cache if they hit and are written to the 60x bus independent of the L2 hit/miss status. If the store hits in the L2 cache, the modified/unmodified status of the tag remains unchanged. All requests to the L2 cache that are marked cache-inhibited by address translation (through either the MMU or by default WIMG configuration) bypass the L2 cache and do not cause any L2 cache tag state change.

The execution of the **stwcx.** instruction results in single-beat writes from the L1 data cache. These single-beat writes are processed by the L2 cache according to hit/miss status, L1 and L2 write-through configuration, and reservation-active status. If the address associated with the **stwcx.** instruction misses in the L2 cache or if the reservation is no longer active, the **stwcx.** instruction bypasses the L2 cache and is forwarded to the 60x bus interface. If the **stwcx.** hits in the L2 cache and the reservation is still active, one of the following actions occurs:

- If the **stwcx.** hits a modified sector in the L2 cache (independent of write-through status), or if the **stwcx.** hits both the L1 and L2 caches in copy-back mode, the **stwcx.** is written to the L2 and the reservation completes.
- If the **stwcx.** hits an unmodified sector in the L2 cache, and either the L1 or L2 is in write-through mode, the **stwcx.** is forwarded to the 60x bus interface and the sector hit in the L2 cache is invalidated.

L1 cache-block-push operations generated by the execution of **dcbf** and **dcbst** instructions write through to the 60x bus interface and invalidate the L2 cache sector if they hit. The execution of **dcbf** and **dcbst** instructions that do not cause a cache-block-push from the L1 cache are forwarded to the L2 cache to perform a sector invalidation and/or push from the L2 cache to the 60x bus as required. If the **dcbf** and **dcbst** instructions do not cause a sector push from the L2 cache, they are forwarded to the 60x bus interface for address-only broadcast if HID0[ABE] is set to 1.

The L2 flush mechanism is similar to the L1 data cache flush mechanism. L2 flush requires that the entire L1 data cache be flushed prior to flushing the L2 cache. Also, interrupts must be disabled during the L2 flush so that the LRU algorithm does not get disturbed. The L2 can be flushed by executing uniquely addressed load instructions to each of the 32 byte blocks of the L2 cache. This requires a load to each of the 2 sets (2-way set associative) of the 32-byte block (sector) within each 64-byte line of the L2 cache. The loads must not hit in the L1 cache in order to effect a flush of the L2 cache.

The **dcbi** instruction is always forwarded to the L2 cache and causes a segment invalidation if a hit occurs. The instruction is also forwarded to the 60x bus interface for broadcast if HID0[ABE] is set to 1. The **icbi** instruction invalidates only L1 cache blocks and is never forwarded to the L2 cache.

Any **dcbz** instructions marked global do not affect the L2 cache state. If an instruction hits in the L1 and L2 caches, the L1 data cache block is cleared and the instruction completes. If an instruction misses in the L2 cache, it is forwarded to the 60x bus interface for broadcast. Any **dcbz** instructions

that are marked nonglobal act only on the L1 data cache without reference to the state of the L2.
The **dcbz_l** is not forwarded to the L2 cache.
The **sync** and **eieio** instructions bypass the L2 cache and are forwarded to the 60x bus.

9.1.2 L2 Cache Control Register (L2CR)

The L2 cache control register is used to configure and enable the L2 cache. The L2CR is a supervisor-level read/write, implementation-specific register that is accessed as SPR 1017. The contents of the L2CR are cleared during power-on reset. Table 9-1 describes the L2CR bits. For additional information about the configuration of the L2CR, refer to Section 2.1.2.11 on Page 2-25.

Table 9-1. L2 Cache Control Register

Bit	Name	Function
0	L2E	L2 enable
1	L2CE	L2 double bit error checkstop enable.
2-8		Reserved.
9	L2DO	L2 data-only. Setting this bit disables the caching of instructions in the L2 cache.
10	L2I	L2 global invalidate. Setting L2I invalidates the L2 cache globally by clearing the L2 status bits.
11		Reserved
12	L2WT	L2 write-through. Setting L2WT selects write-through mode (rather than the default copy-back mode) so all writes to the L2 cache also write through to the 60x bus.
13	L2TS	L2 test support. Setting L2TS causes cache block pushes from the L1 data cache that result from **dcbf** and **dcbst** instructions to be written only into the L2 cache and marked valid, rather than being written only to the 60x bus and marked invalid in the L2 cache in case of hit. If L2TS is set, causes single-beat store operations that miss in the L2 cache to be discarded.
14-30		Reserved.
31	L2IP	L2 global invalidate in progress (read only)—This read-only bit indicates whether an L2 global invalidate is occurring.

9.1.3 L2 Cache Initialization

Following a power-on or hard reset, the L2 cache is disabled initially. Before enabling the L2 cache, other configuration parameters must be set in the L2CR, and the L2 tags must be globally invalidated. The L2 cache should be initialized during system start-up.
The sequence for initializing the L2 cache is:

1. Power-on reset (automatically performed by the assertion of $\overline{\text{HRESET}}$ signal).
2. Disable interrupts and Dynamic Power Management (DPM).
3. Disable L2 cache by clearing L2CR[L2E].
4. Perform an L2 global invalidate as described in the next section.
5. After the L2 global invalidate has been performed, and the other L2 configuration bits have been set, enable the L2 cache for normal operation by setting the L2CR[L2E] bit to 1.

9.1.4 L2 Cache Global Invalidation

The L2 cache supports a global invalidation function in which all bits of the L2 tags (tag data bits, tag status bits, and LRU bit) are cleared. It is performed by an on-chip hardware state machine that sequentially cycles through the L2 tags. The global invalidation function is controlled through L2CR[L2I], and it must be performed only while the L2 cache is disabled. Gekko can continue operation during a global invalidation provided the L2 cache has been properly disabled before the global invalidation operation starts.

The sequence for performing a global invalidation of the L2 cache is as follows:

1. Execute a **sync** instruction to finish any pending store operations in the load/store unit, disable the L2 cache by clearing L2CR[L2E], and execute an additional **sync** instruction after disabling the L2 cache to ensure that any pending operations in the L2 cache unit have completed.
2. Initiate the global invalidation operation by setting the L2CR[L2I] bit to 1.
3. Monitor the L2CR[L2IP] bit to determine when the global invalidation operation is completed (indicated by the clearing of L2CR[L2IP]). The global invalidation requires approximately 32K core clock cycles to complete.
4. After detecting the clearing of L2CR[L2IP], clear L2CR[L2I] and re-enable the L2 cache for normal operation by setting L2CR[L2E].

9.1.5 L2 Cache Test Features and Methods

In the course of system power-up, testing may be required to verify the proper operation of the L2 tag memory, SRAM, and overall L2 cache system. The following sections describe Gekko's features and methods for testing the L2 cache. The L2 cache address space should be marked as guarded (G = 1) so spurious load operations are not forwarded to the 60x bus interface before branch resolution during L2 cache testing.

9.1.5.1 L2CR Support for L2 Cache Testing

L2CR[DO] and L2CR[TS] support the testing of the L2 cache. L2CR[DO] prevents instructions from being cached in the L2. This allows the L1 instruction cache to remain enabled during the testing process without having L1 instruction misses affect the contents of the L2 cache and allows all L2 cache activity to be controlled by program-specified load and store operations.

L2CR[TS] is used with the **dcbf** and **dcbst** instructions to push data into the L2 cache. When L2CR[TS] is set, and the L1 data cache is enabled, an instruction loop containing a **dcbf** instruction can be used to store any address or data pattern to the L2 cache. Additionally, 60x bus broadcasting is inhibited when a **dcbz** instruction is executed. This allows the use of a **dcbz** instruction to clear an L1 cache block, followed by a **dcbf** instruction to push the cache block into the L2 cache and invalidate the L1 cache block.

When the L2 cache is enabled, cacheable single-beat read operations are allowed to hit in the L2 cache and cacheable write operations are allowed to modify the contents of the L2 cache when a hit occurs. Cacheable single-beat read and writes occur when address translation is disabled (invoking the use of the default WIMG bits (0b0011)), or when address translation is enabled and accesses are marked as cacheable through the page table entries or the BATs, and the L1 data cache is disabled or locked. When the L2 cache has been initialized and the L1 cache has been disabled or locked, load or store instructions then bypass the L1 cache and hit in the L2 cache directly. When L2CR[TS] is set, cacheable single-beat writes are inhibited from accessing the 60x bus interface after an L2 cache miss.

During L2 cache testing, the performance monitor can be used to count L2 cache hits and misses, thereby providing a numerical signature for test routines and a way to verify proper L2 cache

operation.

9.1.5.2 L2 Cache Testing

A typical test for verifying the proper operation of Gekko's L2 cache memory would perform the following steps:

1. Initialize the L2 test sequence by disabling address translation to invoke the default WIMG setting (0b0011). Set L2CR[DO] and L2CR[TS] and perform a global invalidation of the L1 data cache and the L2 cache. The L1 instruction cache can remain enabled to improve execution efficiency.
2. Test the L2 cache SRAM by enabling the L1 data cache and executing a sequence of **dcbz**, **stw**, and **dcbf** instructions to initialize the L2 cache with a desired range of consecutive addresses and with cache data consisting of zeros. Once the L2 cache holds a sequential range of addresses, disable the L1 data cache and execute a series of single-beat load and store operations employing a variety of bit patterns to test for stuck bits and pattern sensitivities in the L2 cache SRAM. The performance monitor can be used to verify whether the number of L2 cache hits or misses corresponds to the tests performed.
3. Test the L2 cache tag memory by enabling the L1 data cache and executing a sequence of **dcbz**, **stw**, and **dcbf** instructions to initialize the L2 cache with a wide range of addresses and cache data. Once the L2 cache is populated with a known range of addresses and data, disable the L1 data cache and execute a series of store operations to addresses not previously in the L2 cache. These store operations should miss in every case. Note that setting the L2CR[TS] inhibits L2 cache misses from being forwarded to the 60x bus interface, thereby avoiding the potential for bus errors due to addressing hardware or nonexistent memory. The L2 cache then can be further verified by reading the previously loaded addresses and observing whether all the tags hit, and that the associated data compares correctly. The performance monitor can also be used to verify whether the proper number of L2 cache hits and misses correspond to the test operations performed.
4. The entire L2 cache can be tested by clearing L2CR[DO] and L2CR[TS], restoring the L1 and L2 caches to their normal operational state, and executing a comprehensive test program designed to exercise all the caches. The test program should include operations that cause L2 hit, reload, and castout activity that can be subsequently verified through the performance monitor.

9.1.6 L2 Cache Timing

There is a 64 bit bus to access the L2 SRAM. Access to the L2 data cache typically takes 3 processor cycles. In the first cycle, the address is presented to the data cache and on writes, ECC check bits are generated. The second cycle, the array is accessed. On the third cycle, ECC error correction is done for reads. It takes 4 cycles to transfer the data for each L2 access with "wrap around" to forward the critical double word for read operation.

Accesses to the L2 are not pipelined. Gekko's L2 support

9.2 Locked L1 Data Cache

Under the control of the HID2[LCE] bit, the L1 data cache can be configured as either a 32 Kbyte normal cache, or as a 16 Kbyte normal cache and a 16 Kbyte locked cache. The locked cache can be explicitely managed, separate from the normal cache. A new instruction, **dcbz_l**, is used to allocate cache lines in the locked cache.

9.2.1 Locked Cache Configuration

At power-on or reset, HID2[LCE] is set to be 0. The L1 data cache is a 32 Kbyte 8-way set-associated cache, as described in Chapter 3. When a **mtspr** instruction sets HID2[LCE] = 1, the data cache is configured as two partitions. The first partition, consisting of ways 0-3, is a 16 Kbyte normal cache. The second partition, consisting of ways 4-7, is a 16 Kbyte locked cache. The normal cache operates as described in Chapter 3, except that it behaves as a four-way set-associative cache. The operation of the locked cache partition is described in the following sections.

9.2.2 Locked Cache Operation

The new instruction, **dcbz_l,** is the only mechanism to allocate a tag for a 32 byte block in the locked cache to be associated with a particular address. There are four methods to de-allocate cache lines in the locked cache:

1. Use **dcbi** instruction
2. Use **dcbf** instruction
3. The **dcbz_l** instruction forces cache line replacement by the pseudo-LRU algorithm in the locked cache

The behavior of the cache control instructions are the following:

9.2.2.1 DCBZ

If a **dcbz** instruction misses both the normal cache and the locked cache, then a cache line is allocated from the four ways in the normal cache according to the pseudo-LRU rule. The effect in the L2 and the 60x bus is the same as when HID2[LCE] = 0

If the instruction hits either the locked cache or the normal cache, the cache line is cleared and marked as ‘M’ and the effect in the L2 and the 60x bus is the same as when HID2[LCE] = 0.

9.2.2.2 DCBZ_L

If a **dcbz_l** instruction misses both the normal cache and the locked cache, a cache line is allocated from the four ways in the locked cache according to the pseudo-LRU rule, and the cache line is marked as ‘M’.

If the instruction hits either the normal cache or the locked cache, then the instruction clears all the bytes in the cache line and marks the line as ‘M’.

The **dcbz_l** instruction has no effect on the L2 cache or the 60x bus.

9.2.2.2.1 DCBZ_L Exceptions

The **dcbz_l** instruction causes an alignment exception if the page or the block of the effective address is marked as write-thriugh or cache-inhibited.

The **dcbz_l** instruction is intended to allocate a 32 byte block in the locked cache. When the instruction hits either the normal cache or the locked cache, Gekko sets HID2[DCHERR]=1. In addition, when the situation happens with HID2[DCHEE] = MSR[EE] = MSR[ME] = 1, and HID2[DCHERR] = 0, Gekko also sets SRR1[10]=1 and raises machine check.

When HID2[LCE] = 0, execution of **dcbz_l** causes an illegal instruction exception

9.2.2.3 DCBI

A **dcbi** hit in the locked cache invalidates the cache line and has no effect on L2 or the 60x bus.

A **dcbi** hit in the normal cache invalidates the cache line. The effect on the L2 and the 60x bus is the same as when HID2[LCE] = 0.

9.2.2.4 DCBF

When a **dcbf** hits a modified cache line in either the normal cache or the locked cache, the cache line is castout and is marked ‘I’. The effect on the L2 and the 60x bus is the same as when HID2[LCE] = 0.

9.2.2.5 DCBST

When a **dcbst** hits a modified cache line in either the locked cache or the normal cache, the cache line is castout and the cacheline is marked as ‘E’. The effect on the L2 and the 60x bus is the same as when HID2[LCE] = 0.

9.2.2.6 DCBT and DCBTST

If a **dcbt** or **dcbtst** hits a cache line in either the normal cache or the locked cache, the instruction is treated as a no-op. If the instruction misses both the locked cache and the normal cache, the corresponding cache line is loaded from the external memory to the normal cache the same as the case when HID2[LCE] = 0.

9.2.2.7 Load and Store

Load and store instructions which miss both the locked and the normal caches will result in a cache line load to the normal cache by the pseudo-LRU rule among the four ways in the normal cache.

Load and store instructions which hit either the normal cache or the locked cache will result in the usual MEI state transition and the pseudo-LRU state transition among the four ways in that partition of the cache.

9.3 Direct Memory Access (DMA)

Gekko implements a DMA engine to transfer data between the locked L1 D-cache and the external memory. The DMA engine has a 15-entry FIFO queue for DMA commands and processes the commands sequentially. The DMA engine's operation is controlled by the two special purpose registers: DMAU and DMAL.

9.3.1 DMA Operation

The DMA engine is disabled at power-on with HID2[LCE] = 0. Setting HID2[LCE] = 1 partitions the L1 D-cache and enables the DMA engine. Note that after HID2[LCE] is set to 1, the i-cache must be invalidated prior to executing any **dcbz_l** instructions. Also, for systems which generate snoop transactions, HID2[LCE] shall be kept at 0.

When a **mtspr** instruction sets DMAL[T] = 1 and DMAL[F] = 0, the DMA engine latches the values in DMAU and DMAL to form a DMA command, enqueues the command in the DMA queue and sets DMAL[T] = 0.

HID2[DMAQL] indicates the number of DMA commands in the DMA queue, including the command in progress (if any).

When the DMA queue is not empty, i.e., HID2[DMAQL] != 0, the DMA engine processes the commands sequentially. The starting address of the transfer in the D-cache is DMAL[LC_ADDR] || 0b00000. The starting address of the transfer in the external memory is DMAU[MEM_ADDR] || 0b00000. The number of cache lines to be transfered by a command is DMAU[DMA_LEN_U] || DMAL[DMA_LEN_L], except that a value of zero specifies a length of 128 cache lines. The direction of the transfer is determined by DMAL[LD]. DMAL[LD] = 0 means a transfer from the locked cache to the external memory. DMAL[LD] = 1 means a transfer from the external memory to the locked cache.

For a DMA store command, i.e., DMAL[LD] = 0, the DMA engine performs a D-cache look-up for each of the cache lines sequentially from the starting address. For a look-up hit in the locked cache, the DMA engine initiates a 60x bus write-with-flush transaction to transfer the 32 byte cache line from the locked cache to the external memory.

For a DMA load command, i.e., DMAL[LD] = 1, the DMA engine performs a D-cache look-up for each of the cache lines sequentially from the starting address. For a look-up hit in the locked cache, the DMA engine initiates a 60x bus burst read transaction to transfer the data from the external memory to the locked cache. For all but the last read transaction associated with the DMA load command, the burst read transaction type is 0b01011. The last burst read transaction has a transaction type of 0b01010. Gekko initiates the burst transaction type 0b01011 only for the DMA load commands. The memory controller can use the information to pre-fetch the next cache line to improve the performance.

The DMA access to the cache, either a load or a store, will result in a pseudo-LRU state transition within the four-way set associated with the cache line, but does not affect the MEI state. If the look-up misses the locked cache, the DMA engine transfers no data and continues to the next cache line.

The **eieio** and **sync** instructions have no effect on the DMA engine. When HID0[ABE] = 0, the execution of ***sync*** does not complete until all the DMA commands in the queue are completed. When HID0[ABE] = 1, the execution of **sync** is not affected by the DMA operation.

The only way to flush the DMA queue is to issue a **mtspr** instruction to set DMAL[F] = 1. In this situation, the DMA engine flushes all the commands in the DMA queue, including the command in progress, and sets both DMAL[F] = DMAL[T] = 0. Such an instruction should be followed by a **sync** instruction to ensure that the pending bus transaction associated with the discarded command, if any, complete before the DMA engine accepts the next DMA command.

9.3.2 Exception Conditions

There are three conditions under which a DMA operation can cause an exception.

9.3.2.1 DMA Queue Overflow

When a **mtspr** instruction sets DMAL[T] = 1 and DMAL[F] = 0 while HID2[DMAQL] = 15, the DMA engine does not latch the DMA command, but sets DMAL[T] = 0 and HID2[DQOERR] = 1. In addition, when the situation happens that HID2[DQOEE] = MSR[EE] = MSR[ME] = 1 and HID2[DQOERR] = 0, Gekko also sets SRR1[10] = 1 and raises machine check.

9.3.2.2 DMA Look-up Hits Normal Cache

When the DMA engine looks up the L1 cache tag and hits in the normal cache partition, Gekko transfers no data, continues to the next cache line and indicates the situation by setting HID2[DNCERR] = 1. In addition, when the situation happens that HID2[DNCEE] = MSR[EE] = MSR[ME] = 1 and HID2[DNCERR] = 0, Gekko also sets SRR1[10] =1 and raises machine check.

9.3.2.3 DMA Look-up Miss

When a DMA engine look-up misses the L1 cache tag, Gekko transfers no data, continues to the next cache line and indicates the situation by setting HID2[DCMERR] = 1. In addition, when the situation happens that HID2[DCMEE] = MSR[EE] = MSR[ME] = 1 and HID2[DCMERR] = 0, Gekko also sets SRR1[10] = 1 and raises machine check.

9.3.3 DMA Timing

A DMA command is broken into a sequence of transaction requests. Each request will transfer a 32 byte block between the locked cache and the external memory. The read/write transaction requests from DMA are served by the BIU. On a first-come-first-serve basis, the BIU serves transaction requests from multiple sources, e.g., DMA read, instruction load, etc.

A DMA transaction request requires one cycle to arbitrate for L1 cache tag access and then one cycle to look up the tag for the cache line. After the tag look-up, the DMA makes the transaction request to the BIU.

For a DMA store, it takes one cycle to fetch the 32 byte block from the cache and make the write transaction request to the BIU to transfer the data to the external memory. There is a two entry DMA store queue to support the pipelined write transactions.

For a DMA load, there is a two entry DMA load queue to support the pipelined read transactions from the external memory. After receiving the 32 byte data from the external memory, it takes one cycle to place the data into the cache.

9.4 Write Gather Pipe

Gekko implements a write gather pipe for efficient transfer of non-cacheable data from the processor to the external memory. The write gather pipe consists of a 128 byte circular FIFO buffer and a special purpose register: Write Pipe Address Register (WPAR). For a non-cacheable store instruction to the address specified in WPAR, the operand will be stored sequentially in the buffer. When there are 32 bytes or more of data in the buffer, the write gather pipe will sequentially transfere the data to the external memory by burst transaction.

9.4.1 WPAR

The write gather pipe address register is a 32-bit special purpose register. WPAR holds the upper 26 bits of the physical address of the write pipe, WPAR[GB_ADDR], and a status bit, WPAR[BNE]. The WPAR controls the operations of the write gather pipe.

9.4.2 Write Gather Pipe Operation

The write gather pipe is disabled at power-on as HID2[WPE] = 0. Setting HID2[WPE] = 1 enables the write gather pipe. The operation is described below.

A **mtspr** to WPAR invalidates the data in the buffer and sets the gather address. In other words, all the data in the buffer, yet to be transfered, will be discarded and the operand of all the store instructions to the non-cacheable address of WPAR[GB_ADDR] || 0b00000 will be stored in the buffer.

When there are 32 bytes or more of data stored in the buffer, the write gather pipe will transfer the data to the memory 32 bytes at a time with a write-with-flush burst transaction. The address of the transaction is WPAR[GB_ADDR] || 0b00000. Software can check WPAR[BNE] to determine if the buffer is empty or not.

The **eieio, stwcx**, and **sync** instructions have no effect on the write gather pipe. The write gather pipe does not participate in bus snoop operation. The only way for software to flush out a partially full 32 byte block is to fill up the block with dummy data,. This fill data must be recognized or ignored by the consumer of the data stream to ensure the system's proper behavior.

A non-cacheable store to an address with bits 0-26 matching WPAR[GB_ADDR] but with bits 27-31 not all zero will result in incorrect data in the buffer.

9.4.3 Write Gather Pipe Timing

The buffer of the write gather pipe has independent read and write ports such that the burst transfer does not block the store instructions. However, when the buffer has more than 120 bytes of data pending to be transfered, a non-cacheable store instruction to the gather address stalls.

The cycle following a store to the write gather pipe such that the buffer conatain at least 32 bytes, a transaction request is made to the BIU to burst out 32 bytes of data. As soon as the write transaction request is being served by the BIU, a second write transaction request can be made to the BIU, if an additional 32 bytes has been gathered. On a first-come-first-serve basis, the BIU serves transaction requests from multiple sources, e.g., DMA write, instruction load, etc.

Chapter 10 Power and Thermal Management

Gekko microprocessor is specifically designed for low-power operation. It provides both automatic and program-controlled power reduction modes for progressive reduction of power consumption. It also provides a thermal assist unit (TAU) to allow on-chip thermal measurement, allowing sophisticated thermal management for high-performance portable systems. This chapter describes the hardware support provided by Gekko for power and thermal management.

10.1 Dynamic Power Management

NOTE: The Gekko processor ignores HID0[DPM] and does not implement dynamic power management.

Dynamic power management (DPM) automatically powers up and down the individual execution units of Gekko, based upon the contents of the instruction stream. For example, if no floating-point instructions are being executed, the floating-point unit is automatically powered down. Power is not actually removed from the execution unit; instead, each execution unit has an independent clock input, which is automatically controlled on a clock-by-clock basis. Since CMOS circuits consume negligible power when they are not switching, stopping the clock to an execution unit effectively eliminates its power consumption. The operation of DPM is completely transparent to software or any external hardware. Dynamic power management is enabled by setting HID0[DPM] to 1.

10.2 Programmable Power Modes

Gekko provides four programmable power states—full power, doze, nap, and sleep. Software selects these modes by setting one (and only one) of the three power saving mode bits in the HID0 register.

Hardware can enable a power management state through external asynchronous interrupts. Such a hardware interrupt causes the transfer of program flow to interrupt handler code that then invokes the appropriate power saving mode. Gekko also contains a decrementer which allows it to enter the nap or doze mode for a predetermined amount of time and then return to full power operation through a decrementer interrupt.

NOTE: Gekko cannot switch from one power management mode to another without first returning to full-power mode.

The sleep mode disables bus snooping; therefore, a hardware handshake is provided to ensure coherency before Gekko enters this power management mode.

Table 10-1 summarizes the four power states.

Table 10-1. Gekko Microprocessor Programmable Power Modes

PM Mode	Functioning Units	Activation Method	Full-Power Wake Up Method
Full power	All units active	—	—
Doze	• Bus snooping • Data cache as needed • Decrementer timer	Controlled by SW	External asynchronous exceptions* Decrementer interrupt Performance monitor interrupt Thermal management interrupt Hard or soft reset
Nap	• Bus snooping — enabled by deassertion of $\overline{\text{QACK}}$ • Decrementer timer	Controlled by hardware and software	External asynchronous exceptions* Decrementer interrupt Hard or soft reset
Sleep	None	Controlled by hardware and software	External asynchronous exceptions* Hard or soft reset

Note: * Exceptions are referred to as interrupts in the architecture specification.

10.2.1 Power Management Modes

The following sections describe the characteristics of Gekko's power management modes, the requirements for entering and exiting the various modes, and the system capabilities provided by Gekko while the power management modes are active.

10.2.1.1 Full-Power Mode

Full-power mode is selected when the POW bit in MSR is cleared.

- Default state following power-up and $\overline{\text{HRESET}}$
- All functional units are operating at full processor speed at all times.

10.2.1.2 Doze Mode

Doze mode disables most functional units but maintains cache coherency by enabling the bus interface unit and snooping. A snoop hit causes Gekko to enable the data cache, copy the data back to memory, disable the cache, and fully return to the doze state.

- Most functional units disabled
- Bus snooping and time base/decrementer still enabled
- Doze mode sequence
  - — Set doze bit (HID0[8] = 1), clear nap and sleep bits (HID0[9] and HID0[10] = 0)
  - — Gekko enters doze mode after several processor clocks
- Several methods of returning to full-power mode
  - — Assert $\overline{\text{INT}}$, $\overline{\text{MCP}}$, decrementer, performance monitor, machine check, or thermal management interrupts
  - — Assert hard reset or soft reset

- Transition to full-power state takes no more than a few processor cycles
- PLL running and locked to SYSCLK

10.2.1.3 Nap Mode

The nap mode disables Gekko but still maintains the phase-locked loop (PLL), and the time base/decrementer. The time base can be used to restore Gekko to full-power state after a programmed amount of time. To maintain data coherency, bus snooping is disabled for nap and sleep modes through a hardware handshake sequence using the quiesce request ($\overline{\text{QREQ}}$) and quiesce acknowledge ($\overline{\text{QACK}}$) signals. Gekko asserts the $\overline{\text{QREQ}}$ signal to indicate that it is ready to disable bus snooping. When the system has ensured that snooping is no longer necessary, it will assert $\overline{\text{QACK}}$ and Gekko will enter the nap mode. If the system determines that a bus snoop cycle is required, $\overline{\text{QACK}}$ is deasserted to Gekko for at least eight bus clock cycles, and the Gekko will then be able respond to a snoop cycle. Assertion of $\overline{\text{QACK}}$ following the snoop cycle will again disable Gekko's snoop capability. Gekko's power dissipation while in nap mode with $\overline{\text{QACK}}$ deasserted is the same as the power dissipation while in doze mode.

Gekko (2.0 and later) also allows dynamic switching between nap and doze modes to allow the use of nap mode without sacrificing hardware snoop coherency. For this operation, negating $\overline{\text{QACK}}$ at any time for at least 8 bus cycles guarantees that Gekko has transitioned from nap mode to doze mode in order to snoop. Reasserting QACK then allows Gekko to return to nap mode. This sequencing could be used by the system at any time with knowledge of what power management mode, if any, that Gekko is currently in.

- Time base/decrementer still enabled
- Thermal management unit enabled
- Most functional units disabled
- All nonessential input receivers disabled
- Nap mode sequence
  - — Set nap bit (HID0[9] = 1), clear doze and sleep bits (HID0[8] and HID0[10] = 0)
  - — Gekko asserts quiesce request ($\overline{\text{QREQ}}$) signal
  - — System asserts quiesce acknowledge ($\overline{\text{QACK}}$) signal
  - — Gekko enters nap mode after several processor clocks
- Nap mode bus snoop sequence
  - — System deasserts $\overline{\text{QACK}}$ signal for eight or more bus clock cycles
  - — Gekko snoops address tenure(s) on bus
  - — System asserts $\overline{\text{QACK}}$ signal to restore full nap mode
- Several methods of returning to full-power mode
  - — Assert $\overline{\text{INT}}$, $\overline{\text{MCP}}$, machine check, or decrementer interrupts
  - — Assert hard reset or soft reset

- Transition to full-power takes no more than a few processor cycles
- PLL running and locked to SYSCLK.

10.2.1.4 Sleep Mode

Sleep mode consumes the least amount of power of the four modes since all functional units are disabled. To conserve the maximum amount of power, the PLL may be disabled by placing the PLL_CFG signals in the PLL bypass mode, and disabling SYSCLK.

NOTE: Forcing the SYSCLK signal into a static state does not disable Gekko's PLL, which will continue to operate internally at an undefined frequency unless placed in PLL bypass mode.

Due to the fully static design of Gekko, internal processor state is preserved when no internal clock is present. Because the time base and decrementer are disabled while Gekko is in sleep mode, the Gekko's time base contents will have to be updated from an external time base after exiting sleep mode if maintaining an accurate time-of-day is required. Before entering the sleep mode, Gekko asserts the $\overline{\text{QREQ}}$ signal to indicate that it is ready to disable bus snooping.

When the system has ensured that snooping is no longer necessary, it asserts $\overline{\text{QACK}}$ and the Gekko will enter sleep mode.

- All functional units disabled (including bus snooping and time base)
- All nonessential input receivers disabled
  - — Internal clock regenerators disabled
  - — PLL still running (see below)
- Sleep mode sequence
  - — Set sleep bit (HID0[10] = 1), clear doze and nap bits (HID0[8] and HID0[9])
  - — Gekko asserts quiesce request ($\overline{\text{QREQ}}$)
  - — System asserts quiesce acknowledge ($\overline{\text{QACK}}$)
  - — Gekko enters sleep mode after several processor clocks
- Several methods of returning to full-power mode
  - — Assert $\overline{\text{INT}}$ or $\overline{\text{MCP}}$ interrupts
  - — Assert hard reset or soft reset
- PLL and DLL may be disabled and SYSCLK may be removed while in sleep mode
- Return to full-power mode after PLL and SYSCLK are disabled in sleep mode
  - — Enable SYSCLK
  - — Reconfigure PLL into desired processor clock mode
  - — System logic waits for PLL startup and relock time (100 sec)
  - — System logic asserts one of the sleep recovery signals (for example, INT)

10.2.2 Power Management Software Considerations

Since Gekko is a dual-issue processor with out-of-order execution capability, care must be taken in how the power management mode is entered. Furthermore, nap and sleep modes require all outstanding bus operations to be completed before these power management modes are entered. Normally, during system configuration time, one of the power management modes would be selected by setting the appropriate HID0 mode bit. Later on, the power management mode is invoked by setting the MSR[POW] bit. To ensure a clean transition into and out of a power management mode, set the MSR[EE] bit to 1 and execute the following code sequence:

sync

mtmsr[POW = 1]

isync

continue

10.3 Thermal Assist Unit

With the increasing power dissipation of high-performance processors and operating conditions that span a wider range of temperatures than desktop systems, thermal management becomes an essential part of system design to ensure reliable operation of portable systems. One key aspect of thermal management is ensuring that the junction temperature of the microprocessor does not exceed the operating specification. While the case temperature can be measured with an external thermal sensor, the thermal constant from the junction to the case can be large, and accuracy can be a problem. This may lead to lower overall system performance due to the necessary compensation to alleviate measurement deficiencies.

Gekko provides the system designer an efficient means of monitoring junction temperature through the incorporation of an on-chip thermal sensor and programmable control logic to enable a thermal management implementation tightly coupled to the processor for improved performance and reliability.

10.3.1 Thermal Assist Unit Overview

The on-chip thermal assist unit (TAU) is composed of a thermal sensor, a digital-to-analog converter (DAC), a comparator, control logic, and three dedicated SPRs. See Figure 10-1 for a block diagram of the TAU.

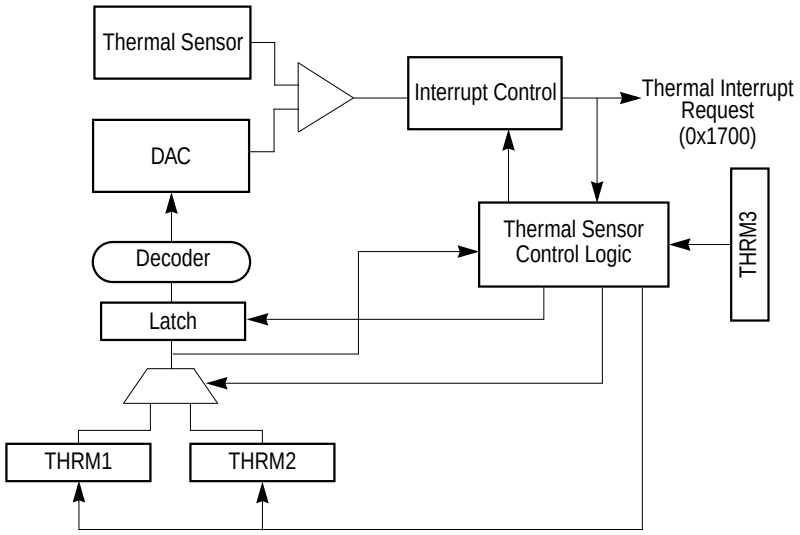


Figure 10-1. Thermal Assist Unit Block Diagram

The TAU provides thermal control by periodically comparing Gekko's junction temperature against user-programmed thresholds, and generating a thermal management interrupt if the threshold values are crossed. The TAU also enables the user to determine the junction temperature through a software successive approximation routine.

The TAU is controlled through three supervisor-level SPRs, accessed through the **mtspr**/**mfspr** instructions. Two of the SPRs (THRM1 and THRM2) provide temperature threshold values that can be compared to the junction temperature value, and control bits that enable comparison and thermal interrupt generation. The third SPR (THRM3) provides a TAU enable bit and a sample interval timer. Note that all the bits in THRM1, THRM2, and THRM3 are cleared to 0 during a hard reset, and the TAU remains idle and in a low-power state until configured and enabled.

The bit fields in the THRM1 and THRM2 SPRs are described in Table 10-2.

Table 10-2. THRM1 and THRM2 Bit Field Settings

Bits	Field	Description
0	TIN	Thermal management interrupt bit. Read only. This bit is set if the thermal sensor output crosses the threshold specified in the SPR. The state of this bit is valid only if TIV is set. The interpretation of the TIN bit is controlled by the TID bit.
1	TIV	Thermal management interrupt valid. Read only. This bit is set by the thermal assist logic to indicate that the thermal management interrupt (TIN) state is valid.
2–8	Threshold	Threshold value that the output of the thermal sensor is compared to. The threshold range is between 0 and 127 C, and each bit represents 1 C. Note that this is not the resolution of the thermal sensor.
9–28	—	Reserved. System software should clear these bits to 0.
29	TID	Thermal management interrupt direction bit. Selects the result of the temperature comparison to set TIN. If TID is cleared to 0, TIN is set and an interrupt occurs if the junction temperature exceeds the threshold. If TID is set to 1, TIN is set and an interrupt is indicated if the junction temperature is below the threshold.
30	TIE	Thermal management interrupt enable. Enables assertion of the thermal management interrupt signal. The thermal management interrupt is maskable by the MSR[EE] bit. If TIE is cleared to 0 and THRM*n* is valid, the TIN bit records the status of the junction temperature vs. threshold comparison without asserting an interrupt signal. This feature allows system software to make a successive approximation to estimate the junction temperature.
31	V	SPR valid bit. This bit is set to indicate that the SPR contains a valid threshold, TID, and TIE controls bits. Setting THRM1/2[V] and THRM3[E] to 1 enables operation of the thermal sensor.

The bit fields in the THRM3 SPR are described in Table 10-3.

Table 10-3. THRM3 Bit Field Settings

Bits	Name	Description
0–17	—	Reserved for future use. System software should clear these bits to 0.
18–30	SITV	Sample interval timer value. Number of elapsed processor clock cycles before a junction temperature vs. threshold comparison result is sampled for TIN bit setting and interrupt generation. This is necessary due to the thermal sensor, DAC, and the analog comparator settling time being greater than the processor cycle time. The value should be configured to allow a sampling interval of 20 microseconds.
31	E	Enables the thermal sensor compare operation if either THRM1[V] or THRM2[V] is set to 1.

10.3.2 Thermal Assist Unit Operation

The TAU can be programmed to operate in single or dual threshold modes, which results in the TAU generating a thermal management interrupt when one or both threshold values are crossed. In addition, with the appropriate software routine, the TAU can also directly determine the junction temperature. The following sections describe the configuration of the TAU to support these modes of operation.

10.3.2.1 TAU Single Threshold Mode

When the TAU is configured for single threshold mode, either THRM1 or THRM2 can be used to contain the threshold value, and a thermal management interrupt is generated when the threshold value is crossed. To configure the TAU for single threshold operation, set the desired temperature threshold, TID, TIE, and V bits for either THRM1 or THRM2. The unused THRM*n* threshold SPR should be disabled by clearing the V bit to 0. In this discussion THRM*n* refers to the THRM threshold SPR (THRM1 or THRM2) selected to contain the active threshold value.

After setting the desired operational parameters, the TAU is enabled by setting the THRM3[E] bit to 1, and placing a value allowing a sample interval of 20 microseconds or greater in the THRM3[SITV] field. The THRM3[SITV] setting determines the number of processor clock cycles between input to the DAC and sampling of the comparator output; accordingly, the use of a value smaller than recommended in the THRM3[SITV] field can cause inaccuracies in the sensed temperature.

If the junction temperature does not cross the programmed threshold, the THRM*n*[TIN] bit is cleared to 0 to indicate that no interrupt is required, and the THRM*n*[TIV] bit is set to 1 to indicate that the TIN bit state is valid. If the threshold value has been crossed, the THRM*n*[TIN] and THRM*n*[TIV] bits are set to 1, and a thermal management interrupt is generated if both the THRM*n*[TIE] and MSR[EE] bits are set to 1.

A thermal management interrupt is held asserted internally until recognized by Gekko's interrupt unit. Once a thermal management interrupt is recognized, further temperature sampling is suspended, and the THRM*n*[TIN] and THRM*n*[TIV] values are held until an **mtspr** instruction is executed to THRM*n*.

The execution of an **mtspr** instruction to THRM*n* anytime during TAU operation will clear the THRM*n*[TIV] bit to 0 and restart the temperature comparison. Executing an **mtspr** instruction to THRM3 will clear both THRM1[TIV] and THRM2[TIV] bits to 0, and restart temperature comparison in THRM*n* if the THRM3[E] bit is set to 1.

Examples of valid THRM1 and THRM2 bit settings are shown in Table 10-4.

Table 10-4. Valid THRM1 and THRM2 Bit Settings

TIN[1]	TIV[1]	TID	TIE	V	Description
x	x	x	x	0	The threshold in the SPR will not be used for comparison.
x	x	x	0	1	Threshold is used for comparison, thermal management interrupt assertion is disabled.
x	x	0	0	1	Set TIN and do not assert thermal management interrupt if the junction temperature exceeds the threshold.
x	x	0	1	1	Set TIN and assert thermal management interrupt if the junction temperature exceeds the threshold.
x	x	1	0	1	Set TIN and do not assert thermal management interrupt if the junction temperature is less than the threshold.
x	x	1	1	1	Set TIN and assert thermal management interrupt if the junction temperature is less than the threshold.
x	0	x	x	1	The state of the TIN bit is not valid.
0	1	0	x	1	The junction temperature is less than the threshold and as a result the thermal management interrupt is not generated for TIE = 1.
1	1	0	x	1	The junction temperature is greater than the threshold and as a result the thermal management interrupt is generated if TIE = 1.
0	1	1	x	1	The junction temperature is greater than the threshold and as a result the thermal management interrupt is not generated for TIE = 1.

Table 10-4. Valid THRM1 and THRM2 Bit Settings (Continued)

TIN[1]	TIV[1]	TID	TIE	V	Description
1	1	1	x	1	The junction temperature is less than the threshold and as a result the thermal management interrupt is generated if TIE = 1.
Note: [1]The TIN and TIV bits are read-only status bits.					

10.3.2.2 TAU Dual-Threshold Mode

The configuration and operation of the TAU's dual-threshold mode is similar to single threshold mode, except both THRM1 and THRM2 are configured with desired threshold and TID values, and the TIE and V bits are set to 1. When the THRM3[E] bit is set to 1 to enable temperature measurement and comparison, the first comparison is made with THRM1. If no thermal management interrupt results from the comparison, the number of processor cycles specified in THRM3[SITV] elapses, and the next comparison is made with THRM2. If no thermal management interrupt results from the THRM2 comparison, the time specified by THRM3[SITV] again elapses, and the comparison returns to THRM1.

This sequence of comparisons continues until a thermal management interrupt occurs, or the TAU is disabled. When a comparison results in an interrupt, the comparison with the threshold SPR causing the interrupt is halted, but comparisons continue with the other threshold SPR. Following a thermal management interrupt, the interrupt service routine must read both THRM1 and THRM2 to determine which threshold was crossed. Note that it is possible for both threshold values to have been crossed, in which case the TAU ceases making temperature comparisons until an **mtspr** instruction is executed to one or both of the threshold SPRs.

10.3.2.3 Gekko Junction Temperature Determination

While Gekko's TAU does not implement an analog-to-digital converter to enable the direct determination of the junction temperature, system software can execute a simple successive approximation routine to find the junction temperature.

The TAU configuration used to approximate the junction temperature is the same required for single-threshold mode, except that the threshold SPR selected has its TIE bit cleared to 0 to disable thermal management interrupt generation. Once the TAU is enabled, the successive approximation routine loads a threshold value into the active threshold SPR, and then continuously polls the threshold SPRs TIV bit until it is set to 1, indicating a valid TIN bit. The successive approximation routine can then evaluate the TIN bit value, and then increment or decrement the threshold value for another comparison. This process is continued until the junction temperature is determined.

10.3.2.4 Power Saving Modes and TAU Operation

The static power saving modes provided by Gekko (the nap, doze, and sleep modes) allow the temperature of the processor to be lowered quickly, and can be invoked through the use of the TAU and associated thermal management interrupt. The TAU remains operational in the nap and doze modes, and in sleep mode as long as the SYSCLK signal input remains active. If the SYSCLK signal is made static when sleep mode is invoked, the TAU is rendered inactive. If Gekko is entering sleep mode with SYSCLK disabled, the TAU should be configured to disable thermal management interrupts to avoid an unwanted thermal management interrupt when the SYSCLK input signal is restored.

10.4 Instruction Cache Throttling

NOTE: Gekko does not implement dynamic power management (DPM), and therefore does not provide thermal reduction through instruction cache throttling.

Gekko provides an instruction cache throttling mechanism to effectively reduce the instruction execution rate without the complexity and overhead of dynamic clock control. Instruction cache throttling, when used in conjunction with the TAU and the dynamic power management capability, provides the system designer with a flexible means of controlling device temperature while allowing the processor to continue operating.

The instruction cache throttling mechanism simply reduces the instruction forwarding rate from the instruction cache to the instruction dispatcher. Normally, the instruction cache forwards four instructions to the instruction dispatcher every clock cycle if all the instructions hit in the cache. For thermal management Gekko provides a supervisor-level instruction cache throttling control (ICTC) SPR. The instruction forwarding rate is reduced by writing a nonzero value into the ICTC[FI] field, and enabling instruction cache throttling by setting the ICTC[E] bit to 1. An overall junction temperature reduction can result in processors that implement dynamic power management by reducing the power to the execution units while waiting for instructions to be forwarded from the instruction cache; thus, instruction cache throttling does not provide thermal reduction unless HID0[DPM] is set to 1.

NOTE: During instruction cache throttling the configuration of the PLL remains unchanged.

The bit field settings of the ICTC SPR are shown in Table 10-5.

Table 10-5. ICTC Bit Field Settings

Bits	Name	Description
23–30	FI	Instruction forwarding interval expressed in processor clocks. 0x00—0 clock cycle 0x01—1 clock cycle : 0xFF—255 clock cycles
31	E	Cache throttling enable 0 Disable instruction cache throttling. 1 Enable instruction cache throttling.

Chapter 11 Performance Monitor

The performance monitor facility provides the ability to monitor and count predefined events such as processor clocks, misses in the instruction cache, data cache, or L2 cache, types of instructions dispatched, mispredicted branches, and other occurrences. The count of such events (which may be an approximation) can be used to trigger the performance monitor exception. The performance monitor facility is not defined by the PowerPC architecture.

The performance monitor can be used for the following:

- To increase system performance with efficient software, especially in a multiprocessing system. Memory hierarchy behavior may be monitored and studied in order to develop algorithms that schedule tasks (and perhaps partition them) and that structure and distribute data optimally.
- To improve processor architecture, the detailed behavior of the PowerPC Gekko's structure must be known and understood in many software environments. Some environments may not be easily characterized by a benchmark or trace.
- To help system developers bring up and debug their systems.

The performance monitor uses the following Gekko-specific special-purpose registers (SPRs):

- The performance monitor counter registers (PMC1–PMC4) are used to record the number of times a certain event has occurred. UPMC1–UPMC4 provide user-level read access to these registers.
- The monitor mode control registers (MMCR0–MMCR1) are used to enable various performance monitor interrupt functions and select events to count. UMMCR0–UMMCR1 provide user-level read access to these registers.
- The sampled instruction address register (SIA) contains the effective address of an instruction executing at or around the time that the processor signals the performance monitor interrupt condition. USIA provides user-level read access to the SIA.

Four 32-bit counters in Gekko count occurrences of software-selectable events. Two control registers (MMCR0 and MMCR1) are used to control performance monitor operation. The counters and the control registers are supervisor-level SPRs; however, in Gekko, the contents of these registers can be read by user-level software using separate SPRs (UMMCR0 and UMMCR1). Control fields in the MMCR0 and MMCR1 select the events to be counted, can enable a counter overflow to initiate a performance monitor exception, and specify the conditions under which counting is enabled.

As with other PowerPC exceptions, the performance monitor interrupt follows the normal PowerPC exception model with a defined exception vector offset (0x00F00). Its priority is below the external interrupt and above the decrementer interrupt.

11.1 Performance Monitor Interrupt

The performance monitor provides the ability to generate a performance monitor interrupt triggered by a counter overflow condition in one of the performance monitor counter registers (PMC1–PMC4), shown in Figure 11-3. A counter is considered to have overflowed when its most-significant bit is set. A performance monitor interrupt may also be caused by the flipping from 0 to 1 of certain bits in the time base register, which provides a way to generate a time reference-based interrupt.

Although the interrupt signal condition may occur with MSR[EE] = 0, the actual exception cannot

be taken until MSR[EE] = 1.

As a result of a performance monitor exception being taken, the action taken depends on the programmable events, as follows: To help track which part of the code was being executed when an exception was signaled, the address of the last completed instruction during that cycle is saved in the SIA. The SIA is not updated if no instruction completed the cycle in which the exception was taken.

Exception handling for the Performance Monitor Interrupt Exception is described in Section 4.5.13, "Performance Monitor Interrupt (0x00F00)" on Page 4-20.

11.2 Special-Purpose Registers Used by Performance Monitor

The performance monitor incorporates the SPRs listed in Table 11-1. All of these supervisor-level registers are accessed through **mtspr** and **mfspr** instructions. The following table shows more information about all performance monitor SPRs.

Table 11-1. Performance Monitor SPRs

SPR Number	spr[5-9] || spr[0-4]	Register Name	Access Level
952	0b11101 11000	MMCR0	Supervisor
953	0b11101 11001	PMC1	Supervisor
954	0b11101 11010	PMC2	Supervisor
955	0b11101 11011	SIA	Supervisor
956	0b11101 11100	MMCR1	Supervisor
957	0b11101 11101	PMC3	Supervisor
958	0b11101 11110	PMC4	Supervisor
936	0b11101 01000	UMMCR0	User (read only)
937	0b11101 01001	UPMC1	User (read only)
938	0b11101 01010	UPMC2	User (read only)
939	0b11101 01011	USIA	User (read only)
940	0b11101 01100	UMMCR1	User (read only)
941	0b11101 01101	UPMC3	User (read only)
942	0b11101 01110	UPMC4	User (read only)

11.2.1 Performance Monitor Registers

This section describes the registers used by the performance monitor.

11.2.1.1 Monitor Mode Control Register 0 (MMCR0)

The monitor mode control register 0 (MMCR0), shown in Figure 11-1, is a 32-bit SPR provided to specify events to be counted and recorded. MMCR0 can be written to only in supervisor mode. User-level software can read the contents of MMCR0 by issuing an **mfspr** instruction to UMMCR0, described in Section 11.2.1.2, "User Monitor Mode Control Register 0 (UMMCR0)" on Page 11-4.

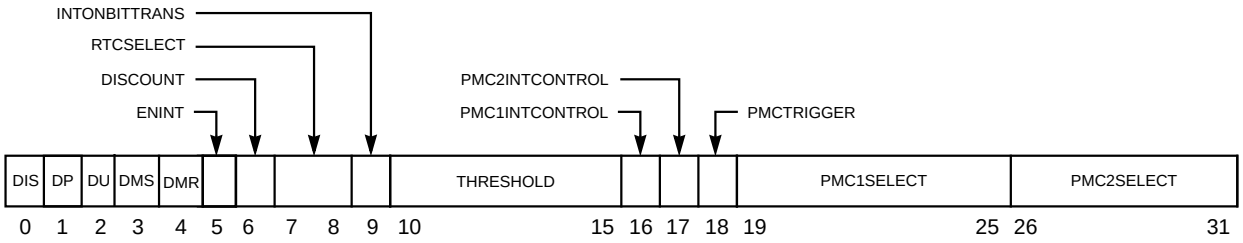


Figure 11-1. Monitor Mode Control Register 0 (MMCR0)

This register must be cleared at power up. Reading this register does not change its contents. Table 11-2 describes the bits of the MMCR0 register.

Table 11-2. MMCR0 Bit Settings

Bit	Name	Description
0	DIS	Disables counting unconditionally. 0 The values of the PMC*n* counters can be changed by hardware. 1 The values of the PMC*n* counters cannot be changed by hardware.
1	DP	Disables counting while in supervisor mode. 0 The PMC*n* counters can be changed by hardware. 1 If the processor is in supervisor mode (MSR[PR] is cleared), the counters are not changed by hardware.
2	DU	Disables counting while in user mode. 0 The PMC*n* counters can be changed by hardware. 1 If the processor is in user mode (MSR[PR] is set), the PMC*n* counters are not changed by hardware.
3	DMS	Disables counting while MSR[PM] is set. 0 The PMC*n* counters can be changed by hardware. 1 If MSR[PM] is set, the PMC*n* counters are not changed by hardware.
4	DMR	Disables counting while MSR[PM] is zero. 0 The PMC*n* counters can be changed by hardware. 1 If MSR[PM] is cleared, the PMC*n* counters are not changed by hardware.
5	ENINT	Enables performance monitor interrupt signaling. 0 Interrupt signaling is disabled. 1 Interrupt signaling is enabled. Cleared by hardware when a performance monitor interrupt is taken. To re-enable these interrupt signals, software must set this bit after servicing the performance monitor interrupt. The IPL ROM code clears this bit before passing control to the operating system.

Table 11-2. MMCR0 Bit Settings (Continued)

Bit	Name	Description
6	DISCOUNT	Disables counting of PMC*n* when a performance monitor interrupt is signaled (that is, ((PMC*n*INTCONTROL = 1) & (PMC*n*[0] = 1) & (ENINT = 1)) or the occurrence of an enabled time base transition with ((INTONBITTRANS =1) & (ENINT = 1)). 0 Signaling a performance monitor interrupt does not affect counting status of PMC*n*. 1 The signaling of a performance monitor interrupt prevents changing of PMC1 counter. The PMC*n* counter does not change if PMC2COUNTCTL = 0. Because a time base signal could have occurred along with an enabled counter overflow condition, software should always reset INTONBITTRANS to zero, if the value in INTONBITTRANS was a one.
7–8	RTCSELECT	Time base lower (TBL) bit selection enable 00 Pick bit 31 to count 01 Pick bit 23 to count 10 Pick bit 19 to count 11 Pick bit 15 to count
9	INTONBITTRANS	Causes interrupt signaling on bit transition (identified in RTCSELECT) from off to on. 0 Do not allow interrupt signal on the transition of a chosen bit. 1 Signal interrupt on the transition of a chosen bit. Software is responsible for setting and clearing INTONBITTRANS.
10–15	THRESHOLD	Threshold value. All 6 bits are supported by Gekko; allowing threshold values from 0 to 63. The intent of the THRESHOLD support is to characterize L1 data cache misses.
16	PMC1INTCONTROL	Enables interrupt signaling due to PMC1 counter overflow. 0 Disable PMC1 interrupt signaling due to PMC1 counter overflow. 1 Enable PMC1 Interrupt signaling due to PMC1 counter overflow.
17	PMCINTCONTROL	Enable interrupt signaling due to any PMC2–PMC4 counter overflow. Overrides the setting of DISCOUNT. 0 Disable PMC2–PMC4 interrupt signaling due to PMC2–PMC4 counter overflow. 1 Enable PMC2–PMC4 interrupt signaling due to PMC2–PMC4 counter overflow.
18	PMCTRIGGER	Can be used to trigger counting of PMC2–PMC4 after PMC1 has overflowed or after a performance monitor interrupt is signaled. 0 Enable PMC2–PMC4 counting. 1 Disable PMC2–PMC4 counting until either PMC1[0] = 1 or a performance monitor interrupt is signaled.
19–25	PMC1SELECT	PMC1 input selector, 128 events selectable; 25 defined. See Table 11-5.
26–31	PMC2SELECT	PMC2 input selector, 64 events selectable; 21 defined. See Table 11-6.

MMCR0 can be accessed with the **mtspr** and **mfspr** instructions using SPR 952.

11.2.1.2 User Monitor Mode Control Register 0 (UMMCR0)

The contents of MMCR0 are reflected to UMMCR0, which can be read by user-level software. UMMCR0 can be accessed with the **mfspr** instructions using SPR 936.

11.2.1.3 Monitor Mode Control Register 1 (MMCR1)

The monitor mode control register 1 (MMCR1) functions as an event selector for performance monitor counter registers 3 and 4 (PMC3 and PMC4). The MMCR1 register is shown in Figure 11-2.

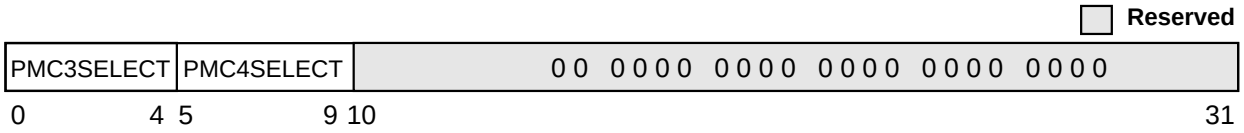


Figure 11-2. Monitor Mode Control Register 1 (MMCR1)

Bit settings for MMCR1 are shown in Table 11-3. The corresponding events are described in Section 11.2.1.5.

Table 11-3. MMCR1 Bit Settings

Bits	Name	Description
0–4	PMC3SELECT	PMC3 input selector. 32 events selectable. See Table 11-7 for defined selections.
5–9	PMC4SELECT	PMC4 input selector. 32 events selectable. See Table 11-8 for defined selections.
10–31	—	Reserved

MMCR1 can be accessed with the **mtspr** and **mfspr** instructions using SPR 956. User-level software can read the contents of MMCR1 by issuing an **mfspr** instruction to UMMCR1, described in Section 11.2.1.4

11.2.1.4 User Monitor Mode Control Register 1 (UMMCR1)

The contents of MMCR1 are reflected to UMMCR1, which can be read by user-level software. UMMCR1 can be accessed with the **mfspr** instructions using SPR 940.

11.2.1.5 Performance Monitor Counter Registers (PMC1–PMC4)

PMC1–PMC4, shown in Figure 11-3, are 32-bit counters that can be programmed to generate interrupt signals when they overflow.

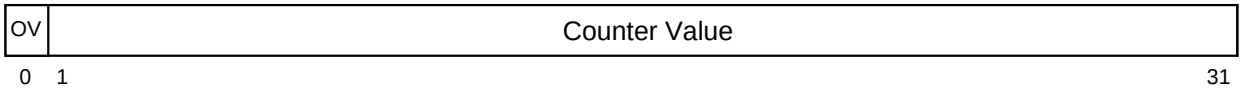


Figure 11-3. Performance Monitor Counter Registers (PMC1–PMC4)

The bits contained in the PMC registers are described in Table 11-4.

Table 11-4. PMC*n* Bit Settings

Bits	Name	Description
0	OV	Overflow. When this bit is set, it indicates this counter has reached its maximum value.
1–31	Counter value	Indicates the number of occurrences of the specified event.

Counters overflow when the high-order bit (the sign bit) becomes set; that is, they reach the value 2147483648 (0x8000_0000). However, an interrupt is not signaled unless both MMCR0[ENINT] and either PMC1INTCONTROL or PMCINTCONTROL in the MMCR0 register are also set as appropriate.

NOTE: The interrupts can be masked by clearing MSR[EE]; the interrupt signal condition may occur with MSR[EE] cleared, but the exception is not taken until MSR[EE] is set. Setting MMCR0[DISCOUNT] forces counters to stop counting when a counter interrupt occurs.

Software is expected to use the **mtspr** instruction to explicitly set PMC to non-overflowed values. Setting an overflowed value may cause an erroneous exception. For example, if both MMCR0[ENINT] and either PMC1INTCONTROL or PMCINTCONTROL are set and the **mtspr** instruction loads an overflow value, an interrupt signal may be generated without an event counting having taken place.

The event to be monitored can be chosen by setting MMCR0[19–31]. The selected events are counted beginning when MMCR0 is set until either MMCR0 is reset or a performance monitor interrupt is generated. Table 11-5 lists the selectable events and their encodings.

Table 11-5. PMC1 Events—MMCR0[19–25] Select Encodings

Encoding	Description
000 0000	Register holds current value.
000 0001	Number of processor cycles
000 0010	Number of instructions that have completed. Does not include folded branches.
0000011	Number of transitions from 0 to 1 of specified bits in the time base lower (TBL) register. Bits are specified through RTCSELECT, MMCR0[7–8]. 00 = 31, 01 = 23, 10 = 19, 11 = 15
0000100	Number of instructions dispatched—0, 1, or 2 instructions per cycle
0000101	Number of **eieio** instructions completed
0000110	Number of cycles spent performing table search operations for the ITLB
0000111	Number of accesses that hit the L2. This event includes cache ops (i.e., dcbz)
0001000	Number of valid instruction EAs delivered to the memory subsystem
0001001	Number of times the address of an instruction being completed matches the address in the IABR
0001010	Number of loads that miss the L1 with latencies that exceeded the threshold value
0001011	Number of branches that are unresolved when processed
0001100	Number of cycles the dispatcher stalls due to a second unresolved branch in the instruction stream
All others	Reserved. May be used in a later revision.

Bits MMCR0[26–31] specify events associated with PMC2, as shown in Table 11-6.

Table 11-6. PMC2 Events—MMCR0[26–31] Select Encodings

Encoding		Description
00 0000	Nothing	Register holds current value.
00 0001	Processor cycles	Count every cycle
00 0010	Number of instructions that have completed.	Indicates number of instructions that have completed. Does not include folded branches
00 0011	Time-base (lower) bit transitions.	Number of transitions from 0 to 1 of specified bits in the time base lower (TBL) register. Bits are specified through RTCSELECT, MMCR0[7–8]. 00 = 31, 01 = 23, 10 = 19, 11 = 15
00 0100	Number of instructions dispatched.	0, 1, or 2 instructions per cycle
00 0101	Number of L1 Icache misses	Indicates the number of times an instruction fetch missed the L1 instruction cache.
00 0110	Number of ITLB misses	Indicates the number of times the needed instruction address translation was not in the ITLB.
00 0111	L2 I-misses	Counts the number of accesses which miss the L2 due to an I-side request.
00 1000	Number of fall-through branches	Indicates the number of branches that were predicted not taken.
00 1001	Reserved.	-
00 1010	Reserved loads	Incremented every time that a reserved load completes.
00 1011	Loads and stores	Counts all load and store instructions completed.
00 1100	Number of snoops	Gives the total number of snoops to the L1 and the L2.
001101	L1 castouts to L2	Number of times the L1 castout goes to the L2.
001110	System Unit Instructions	Number of system unit instructions completed.
001111	Instruction Miss cycles	Counts the total number of L1 miss cycles of instruction fetches.
010000	First speculative branch resolved correctly	Indicates the number of branches that allow speculative execution beyond those that resolved correctly
All others	Reserved.	May be used in a later revision.

Bits MMCR1[0–4] specify events associated with PMC3, as shown in Table 11-7.

Table 11-7. PMC3 Events—MMCR1[0–4] Select Encodings

Encoding	Description
0 0000	Register holds current value.
0 0001	Number of processor cycles
0 0010	Number of completed instructions, not including folded branches.
0 0011	Number of transitions from 0 to 1 of specified bits in the time base lower (TBL) register. Bits are specified through RTCSELECT, MMCR0[7–8]. 00 = 31, 01 = 23, 10 = 19, 11 = 15
0 0100	Number of instructions dispatched. 0, 1, or 2 per cycle.
0 0101	Number of L1 data cache misses. Does not include cache ops.
0 0110	Number of DTLB misses
0 0111	Number of L2 data misses
0 1000	Number of predicted branches that were taken
0 1001	Reserved.
0 1010	Number of store conditional instructions completed
0 1011	Number of instructions completed from the FPU
0 1100	Number of L2 castouts caused by snoops to modified lines
0 1101	Number of cache operations that hit in the L2 cache
0 1110	Reserved
0 1111	Number of cycles generated by L1 load misses
1 0000	Number of branches in the second speculative stream that resolve correctly
1 0001	Number of cycles the BPU stalls due to LR or CR unresolved dependencies
All others	Reserved. May be used in a later revision.

Bits MMCR1[5–9] specify events associated with PMC4, as shown in Table 11-8.

Table 11-8. PMC4 Events—MMCR1[5–9] Select Encodings

Encoding	Comments
00000	Register holds current value
00001	Number of processor cycles
00010	Number of completed instructions, not including folded branches
00011	Number of transitions from 0 to 1 of specified bits in the time base lower (TBL) register. Bits are specified through RTCSELECT, MMCR0[7–8]. 00 = 31, 01 = 23, 10 = 19, 11 = 15
00100	Number of instructions dispatched. 0, 1, or 2 per cycle
00101	Number of L2 castouts
00110	Number of cycles spent performing table searches for DTLB accesses.
00111	Reserved. May be used in a later revision.
01000	Number of mispredicted branches. Reserved for future use.
01001	Reserved. May be used in a later revision.
01010	Number of store conditional instructions completed with reservation intact
01011	Number of completed **sync** instructions
01100	Number of snoop request retries
01101	Number of completed integer operations
01110	Number of cycles the BPU cannot process new branches due to having two unresolved branches
All others	Reserved. May be used in a later revision.

The PMC registers can be accessed with the **mtspr** and **mfspr** instructions using the following SPR numbers:

- PMC1 is SPR 953
- PMC2 is SPR 954
- PMC3 is SPR 957
- PMC4 is SPR 958

11.2.1.6 User Performance Monitor Counter Registers (UPMC1–UPMC4)

The contents of the PMC1–PMC4 are reflected to UPMC1–UPMC4, which can be read by user-level software. The UPMC registers can be read with the **mfspr** instructions using the following SPR numbers:

- UPMC1 is SPR 937
- UPMC2 is SPR 938
- UPMC3 is SPR 941
- UPMC4 is SPR 942

11.2.1.7 Sampled Instruction Address Register (SIA)

The sampled instruction address register (SIA) is a supervisor-level register that contains the

effective address of an instruction executing at or around the time that the processor signals the performance monitor interrupt condition. The SIA is shown in Figure 11-4.

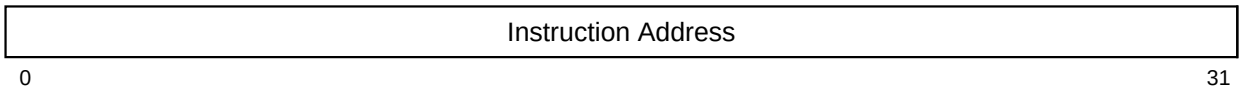


Figure 11-4. Sampled instruction Address Registers (SIA)

If the performance monitor interrupt is triggered by a threshold event, the SIA contains the address of the exact instruction (called the sampled instruction) that caused the counter to overflow.

If the performance monitor interrupt was caused by something besides a threshold event, the SIA contains the address of the last instruction completed during that cycle. SIA can be accessed with the **mtspr** and **mfspr** instructions using SPR 955.

11.2.1.8 User Sampled Instruction Address Register (USIA)

The contents of SIA are reflected to USIA, which can be read by user-level software. USIA can be accessed with the **mfspr** instructions using SPR 939.

11.3 Event Counting

Counting can be enabled if conditions in the processor state match a software-specified condition. Because a software task scheduler may switch a processor's execution among multiple processes and because statistics on only a particular process may be of interest, a facility is provided to mark a process. The performance monitor (PM) bit, MSR[29] is used for this purpose. System software may set this bit when a marked process is running. This enables statistics to be gathered only during the execution of the marked process. The states of MSR[PR] and MSR[PM] together define a state that the processor (supervisor or program) and the process (marked or unmarked) may be in at any time. If this state matches a state specified by the MMCR, the state for which monitoring is enabled, counting is enabled.

The following are states that can be monitored:

- (Supervisor) only
- (User) only
- (Marked and user) only
- (Not marked and user) only
- (Marked and supervisor) only
- (Not marked and supervisor) only
- (Marked) only
- (Not marked) only

In addition, one of two unconditional counting modes may be specified:

- Counting is unconditionally enabled regardless of the states of MSR[PM] and MSR[PR]. This can be accomplished by clearing MMCR0[0–4].
- Counting is unconditionally disabled regardless of the states of MSR[PM] and MSR[PR]. This is done by setting MMCR0[0].

The performance monitor counters count specified events and are used to generate performance monitor exceptions when an overflow (most-significant bit is a 1) situation occurs. The Gekko performance monitor has four, 32-bit registers that can count up to 0x7FFFFFFF (2,147,483,648 in decimal) before overflowing. Bit 0 of the registers is used to determine when an interrupt condition exists.

11.4 Event Selection

Event selection is handled through MMCR0 and MMCR1, described in Table 11-2 on Page 11-3 and Table 11-3 on Page 11-5, respectively. Event selection is described as follows:

- The four event-select fields in MMCR0 and MMCR1 are as follows:
  - MMCR0[19–25] PMC1SELECT—PMC1 input selector, 128 events selectable; 25 defined. See Table 11-5.
  - MMCR0[26–31] PMC2SELECT—PMC2 input selector, 64 events selectable; 21 defined. See Table 11-6.
  - MMCR0[0–4] PMC3SELECT—PMC3 input selector. 32 events selectable, defined. See Table 11-7.
  - MMCR0[5–9] PMC4SELECT—PMC4 input selector. 32 events selectable. See Table 11-8.
- In the tables, a correlation is established between each counter, events to be traced, and the pattern required for the desired selection.
- The first five events are common to all four counters and are considered to be reference events. These are as follows:
  - 00000—Register holds current value
  - 00001—Number of processor cycles
  - 00010—Number of completed instructions, not including folded branches
  - 00011—Number of transitions from 0 to 1 of specified bits in the time base lower (TBL) register. Bits are specified through RTCSELECT, MMCR0[7–8]. 00 = 31, 01 = 23, 10 = 19, 11 = 15
  - 00100—Number of instructions dispatched. 0, 1, or 2 per cycle
- Some events can have multiple occurrences per cycle, and therefore need two or three bits to represent them.

11.5 Notes

The following warnings should be noted:

- Only those load and store in queue position 0 of their respective load/store queues are monitored when a threshold event is selected in PMC1.
- Gekko cannot accurately track threshold events with respect to the following types of loads and stores:
  - Unaligned load and store operations that cross a word boundary
  - Load and store multiple operations
  - Load and store string operations

Chapter 12 Instruction Set

This chapter lists the PowerPC instruction set in alphabetical order by mnemonic. Note that each entry includes the instruction formats and a quick reference ‘legend’ that provides such information as the level(s) of the PowerPC architecture in which the instruction may be found—user instruction set architecture (UISA), virtual environment architecture (VEA), and operating environment architecture (OEA); and the privilege level of the instruction—user- or supervisor-level (an instruction is assumed to be user-level unless the legend specifies that it is supervisor-level); and the instruction formats. The format diagrams show, horizontally, all valid combinations of instruction fields; for a graphical representation of these instruction formats.

A description of the instruction fields and pseudocode conventions are also provided.

NOTE: The architecture specification refers to user-level and supervisor-level as problem state and privileged state, respectively.

12.1 Instruction Formats

Instructions are four bytes long and word-aligned, so when instruction addresses are presented to the processor (as in branch instructions) the two low-order bits are ignored. Similarly, whenever the processor develops an instruction address, its two low-order bits are zero.

Bits 0–5 always specify the primary opcode. Many instructions also have an extended opcode. The remaining bits of the instruction contain one or more fields for the different instruction formats.

Some instruction fields are reserved, or must contain a predefined value as shown in the individual instruction layouts. If a reserved field does not have all bits cleared, or if a field that must contain a particular value does not contain that value, the instruction form is invalid and the results are described in Chapter 4, “Addressing Modes and Instruction Set Summary” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Within the instruction format diagram the instruction operation code and extended operation code (if extended form) are specified in decimal. These fields have been converted to hexadecimal and are shown on line two for each instruction definition.

12.1.1 Split-Field Notation

Some instruction fields occupy more than one contiguous sequence of bits or occupy a contiguous sequence of bits used in permuted order. Such a field is called a split field. Split fields that represent the concatenation of the sequences from left to right are shown in lowercase letters. These split fields— spr and tbr—are described in Table 12-1.

Table 12-1. Split-Field Notation and Conventions

Field	Description
spr (11–20)	This field is used to specify a special-purpose register for the **mtspr** and **mfspr** instructions. The encoding is described in Section 4.4.2.2, “Move to/from Special-Purpose Register Instructions (OEA)”**,** in the *PowerPC Microprocessor Family: The Programming Environments* manual**.**
tbr (11–20)	This field is used to specify either the time base lower (TBL) or time base upper (TBU).

12.1.2 Instruction Fields

Table 12-2 describes the instruction fields used in the various instruction formats.

Table 12-2. Instruction Syntax Conventions

Field	Description
AA (30)	Absolute address bit. 0 The immediate field represents an address relative to the current instruction address (CIA). (For more information on the CIA, see Table 12-3.) The effective (logical) address of the branch is either the sum of the LI field sign-extended to 32 bitsand the address of the branch instruction or the sum of the BD field sign-extended to 32 bits and the address of the branch instruction. 1 The immediate field represents an absolute address. The effective address (EA) of the branch is the LI field sign-extended to 32 bitsor the BD field sign-extended to 32 bits. **Note:** The LI and BD fields are sign-extended to 32 bits.
BD (16–29)	Immediate field specifying a 14-bit signed two's complement branch displacement that is concatenated on the right with 0b00 and sign-extended to 32 bits.
BI (11–15)	This field is used to specify a bit in the CR to be used as the condition of a branch conditional instruction.
BO (6–10)	This field is used to specify options for the branch conditional instructions. The encoding is described in Section 4.2.4.2, “Conditional Branch Control” in the *PowerPC Microprocessor Family: The Programming Environments* manual.
crbA (11–15)	This field is used to specify a bit in the CR to be used as a source.
crbB (16–20)	This field is used to specify a bit in the CR to be used as a source.
crbD (6–10)	This field is used to specify a bit in the CR, or in the FPSCR, as the destination of the result of an instruction.
crfD (6–8)	This field is used to specify one of the CR fields, or one of the FPSCR fields, as a destination.
crfS (11–13)	This field is used to specify one of the CR fields, or one of the FPSCR fields, as a source.
CRM (12–19)	This field mask is used to identify the CR fields that are to be updated by the **mtcrf** instruction.
d (16–31, or 20-31)	Immediate field specifying a signed two's complement integer that is sign-extended to 32 bits.
FM (7–14)	This field mask is used to identify the FPSCR fields that are to be updated by the **mtfsf** instruction.
frA (11–15)	This field is used to specify an FPR as a source.
frB (16–20)	This field is used to specify an FPR as a source.
frC (21–25)	This field is used to specify an FPR as a source.
frD (6–10)	This field is used to specify an FPR as the destination.
frS (6–10)	This field is used to specify an FPR as a source.
I (17-19, or 22-24)	This field is used to specify a GQR control register that is used by the paired single load or store instructions.
IMM (16–19)	Immediate field used as the data to be placed into a field in the FPSCR.
LI (6–29)	Immediate field specifying a 24-bit signed two's complement integer that is concatenated on the right with 0b00 and sign-extended to 32 bits.

Table 12-2. Instruction Syntax Conventions (Continued)

Field	Description
LK (31)	Link bit. 0 Does not update the link register (LR). 1 Updates the LR. If the instruction is a branch instruction, the address of the instruction following the branch instruction is placed into the LR.
MB (21–25) and ME (26–30)	These fields are used in rotate instructions to specify a 32-bit mask in the *PowerPC Microprocessor Family: The Programming Environments* manual.
NB (16–20)	This field is used to specify the number of bytes to move in an immediate string load or store.
OE (21)	This field is used for extended arithmetic to enable setting OV and SO in the XER.
OPCD (0–5)	Primary opcode field
r*A* (11–15)	This field is used to specify a GPR to be used as a source or destination.
r*B* (16–20)	This field is used to specify a GPR to be used as a source.
Rc (31)	Record bit. 0 Does not update the condition register (CR). 1 Updates the CR to reflect the result of the operation. For integer instructions, CR bits 0–2 are set to reflect the result as a signed quantity and CR bit 3 receives a copy of the summary overflow bit, XER[SO]. The result as an unsigned quantity or a bit string can be deduced from the EQ bit. For floating-point instructions, CR bits 4–7 are set to reflect floating-point exception, floating-point enabled exception, floating-point invalid operation exception, and floating-point overflow exception. (Note that exceptions are referred to as interrupts in the architecture specification.)
rD (6–10)	This field is used to specify a GPR to be used as a destination.
rS (6–10)	This field is used to specify a GPR to be used as a source.
SH (16–20)	This field is used to specify a shift amount.
SIMM (16–31)	This immediate field is used to specify a 16-bit signed integer.
SR (12–15)	This field is used to specify one of the 16 segment registers.
TO (6–10)	This field is used to specify the conditions on which to trap. The encoding is described in Section 4.2.4.6, “Trap Instructions” iin the *PowerPC Microprocessor Family: The Programming Environments* manual.
UIMM (16–31)	This immediate field is used to specify a 16-bit unsigned integer.
XO (21–30, 22–30, 25-30 or 26–30)	Extended opcode field.

12.1.3 Notation and Conventions

The operation of some instructions is described by a semiformal language (pseudocode). See Table 12-3 for a list of pseudocode notation and conventions used throughout this chapter

Table 12-3. Notation and Conventions

Notation/Convention	Meaning
←	Assignment
←iea	Assignment of an 32-bit instruction effective address.
¬	NOT logical operator
*	Multiplication
÷	Division (yielding quotient)
+	Two's-complement addition
–	Two's-complement subtraction, unary minus
=,≠	Equals and Not Equals relations
<,≤,≥, >,	Signed comparison relations
. (period)	Update. When used as a character of an instruction mnemonic, a period (.) means that the instruction updates the condition register field.
c	Carry. When used as a character of an instruction mnemonic, a 'c' indicates a carry out in XER[CA].
e	Extended Precision. When used as the last character of an instruction mnemonic, an 'e' indicates the use of XER[CA] as an operand in the instruction and records a carry out in XER[CA].
o	Overflow. When used as a character of an instruction mnemonic, an 'o' indicates the record of an overflow in XER[OV] and CR0[SO] for integer instructions or CR1[SO] for floating-point instructions.
<U, >U	Unsigned comparison relations
?	Unordered comparison relation
&, |	AND, OR logical operators
||	Used to describe the concatenation of two values (that is, 010 || 111 is the same as 010111)
⊕, ≡	Exclusive-OR, Equivalence logical operators (for example, (a ≡ b) = (a ⊕ ¬ b))
0b*nnnn*	A number expressed in binary format.
0x*nnnn or* x'nnnn nnnn'	A number expressed in hexadecimal format.
(*n*)x	The replication of x, *n* times (that is, x concatenated to itself *n* – 1 times). (*n*)0 and (*n*)1 are special cases. A description of the special cases follows: • (*n*)0 means a field of *n* bits with each bit equal to 0. Thus (5)0 is equivalent to 0b00000. • (*n*)1 means a field of *n* bits with each bit equal to 1. Thus (5)1 is equivalent to 0b11111.

Table 12-3. Notation and Conventions (Continued)

Notation/Convention	Meaning
(**r**A|0)	The contents of **r**A if the **r**A field has the value 1–31, or the value 0 if the **r**A field is 0.
(**r**X)	The contents of **r**X
x[*n*]	*n* is a bit or field within x, where x is a register
x^n	x is raised to the *n*th power
ABS(x)	Absolute value of x
CEIL(x)	Least integer ≥ x
Characterization	Reference to the setting of status bits in a standard way that is explained in the text.
CIA	Current instruction address. The 32-bit address of the instruction being described by a sequence of pseudocode. Used by relative branches to set the next instruction address (NIA) and by branch instructions with LK = 1 to set the link register. Does not correspond to any architected register.
Clear	Clear the leftmost or rightmost *n* bits of a register to 0. This operation is used for rotate and shift instructions.
Clear left and shift left	Clear the leftmost *b* bits of a register, then shift the register left by *n* bits. This operation can be used to scale a known non-negative array index by the width of an element. These operations are used for rotate and shift instructions.
Cleared	Bits are set to 0.
Do	Do loop. • Indenting shows range. • “To” and/or “by” clauses specify incrementing an iteration variable. • “While” clauses give termination conditions.
DOUBLE(x)	Result of converting x from floating-point single-precision format to floating-point double-precision format.
Extract	Select a field of *n* bits starting at bit position *b* in the source register, right or left justify this field in the target register, and clear all other bits of the target register to zero. This operation is used for rotate and shift instructions.
EXTS(x)	Result of extending x on the left with sign bits
GPR(x)	General-purpose register x
if...then...else...	Conditional execution, indenting shows range, else is optional.
Insert	Select a field of *n* bits in the source register, insert this field starting at bit position *b* of the target register, and leave other bits of the target register unchanged. (No simplified mnemonic is provided for insertion of a field when operating on double words; such an insertion requires more than one instruction.) This operation is used for rotate and shift instructions. (Note that simplified mnemonics are referred to as extended mnemonics in the architecture specification.)
Leave	Leave innermost do loop, or the do loop described in leave statement.
MASK(x, y)	Mask having ones in positions x through y (wrapping if x > y) and zeros elsewhere.
MEM(x, y)	Contents of y bytes of memory starting at address x

Table 12-3. Notation and Conventions (Continued)

Notation/Convention	Meaning
NIA	Next instruction address, which is the32-bit address of the next instruction to be executed (the branch destination) after a successful branch. In pseudocode, a successful branch is indicated by assigning a value to NIA. For instructions which do not branch, the next instruction address is CIA + 4. Does not correspond to any architected register.
OEA	PowerPC operating environment architecture
Rotate	Rotate the contents of a register right or left *n* bits without masking. This operation is used for rotate and shift instructions.
reserved	
ROTL(x, y)	Result of rotating the value x left y positions, where x is 32 bits long
Set	Bits are set to 1.
Shift	Shift the contents of a register right or left *n* bits, clearing vacated bits (logical shift). This operation is used for rotate and shift instructions.
SINGLE(x)	Result of converting x from floating-point double-precision format to floating-point single-precision format.
SPR(x)	Special-purpose register x
TRAP	Invoke the system trap handler.
Undefined	An undefined value. The value may vary from one implementation to another, and from one execution to another on the same implementation.
UISA	PowerPC user instruction set architecture
VEA	PowerPC virtual environment architecture

Table 12-4 describes instruction field notation conventions used throughout this chapter.

Table 12-4. Instruction Field Conventions

The Architecture Specification	Equivalent to:
BA, BB, BT	**crb**A, **crb**B, **crb**D (respectively)
BF, BFA	**crf**D, **crf**S (respectively)
D	d
DS	ds
FLM	FM
FRA, FRB, FRC, FRT, FRS	**fr**A, **fr**B, **fr**C, **fr**D, **fr**S (respectively)
FXM	CRM
RA, RB, RT, RS	**r**A, **r**B, **r**D, **r**S (respectively)
SI	SIMM

Table 12-4. Instruction Field Conventions (Continued)

The Architecture Specification	Equivalent to:
U	IMM
UI	UIMM
/, //, ///	0...0 (shaded)

Precedence rules for pseudocode operators are summarized in Table 12-5.

Table 12-5. Precedence Rules

Operators	Associativity
x[*n*], function evaluation	Left to right
(*n*)x or replication, x(*n*) or exponentiation	Right to left
unary –, ¬	Right to left
*,	Left to right
+, –	Left to right
||	Left to right
=, , <, , >, , <,U>U, ?	Left to right
&, ⊕, ≡	Left to right
|	Left to right
– (range)	None
←, ←iea	None

Operators higher in Table 12-5 are applied before those lower in the table. Operators at the same level in the table associate from left to right, from right to left, or not at all, as shown. For example, "–" (unary minus) associates from left to right, so a – b – c = (a – b) – c. Parentheses are used to override the evaluation order implied by Table 12-5, or to increase clarity; parenthesized expressions are evaluated before serving as operands.Note that the all pseudocode examples provided in this chapter are for 32-bit implementations.PowerPC Instruction Set

12.1.4 Computation Modes

The PowerPC architecture is defined for 32-bit implementations, in which all registers except the FPRs are 32 bits long, and effective addresses are 32 bits long. The FPR registers are 64 bits long. For more information on computation modes see Section 4.1.1, "Computation Modes," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

12.2 PowerPC Instruction Set

The remainder of this chapter lists and describes the instruction set for the PowerPC architecture. The instructions are listed in alphabetical order by mnemonic. Figure 12-1 shows the format for each instruction description page.

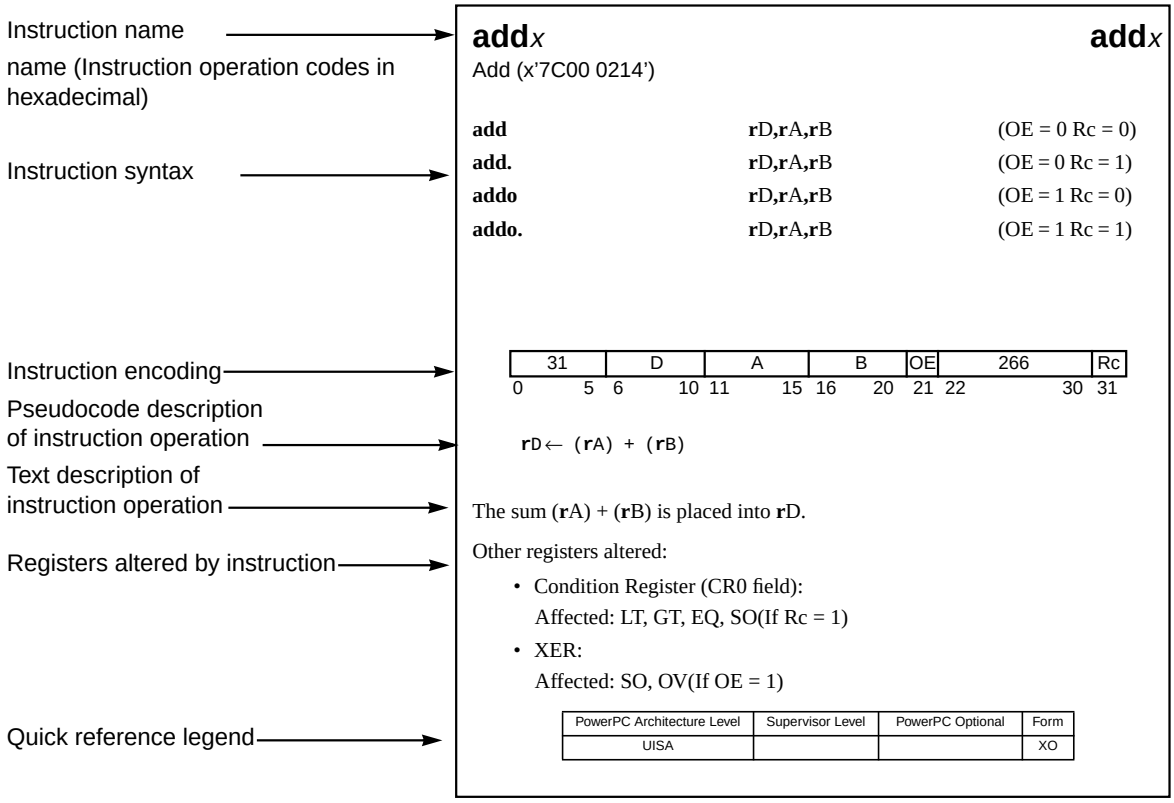


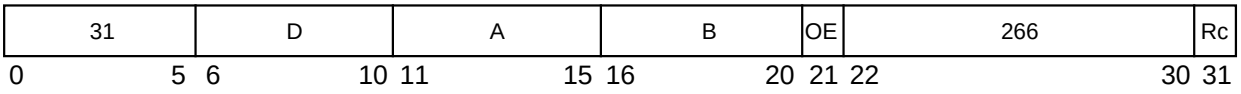
Figure 12-1. Instruction Description

NOTE: The execution unit that executes the instruction may not be the same for all PowerPC processors.

add*x* add*x*

Add (x'7C00 0214')

add **r**D,**r**A,**r**B (OE = 0 Rc = 0)
add. **r**D,**r**A,**r**B (OE = 0 Rc = 1)
addo **r**D,**r**A,**r**B (OE = 1 Rc = 0)
addo. **r**D,**r**A,**r**B (OE = 1 Rc = 1)



rD← (**r**A) + (**r**B)

The sum (**r**A) + (**r**B) is placed into **r**D.

The **add** instruction is preferred for addition because it sets few status bits.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

  NOTE: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).

- XER:

  Affected: SO, OV (if OE = 1)

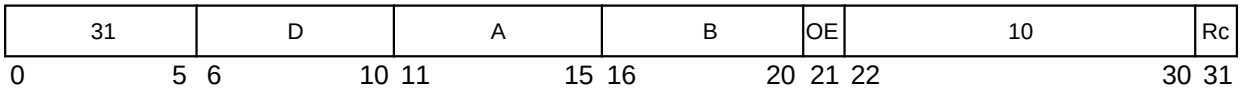
  NOTE: For more information on condition codes see Section 2.1.3, "Condition Register," in the *PowerPC Microprocessor Family: The Programming Environments* manual and Section 2.1.5, "XER Register," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

addc*x* addc*x*

Add Carrying (x’7C00 0014’)

addc **r**D,**r**A,**r**B (OE = 0 Rc = 0)
addc. **r**D,**r**A,**r**B (OE = 0 Rc = 1)
addco **r**D,**r**A,**r**B (OE = 1 Rc = 0)
addco. **r**D,**r**A,**r**B (OE = 1 Rc = 1)



rD← (**r**A) + (**r**B)

The sum (**r**A) + (**r**B) is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

  NOTE: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).

- XER:

  Affected: CA

  Affected: SO, OV (if OE = 1)

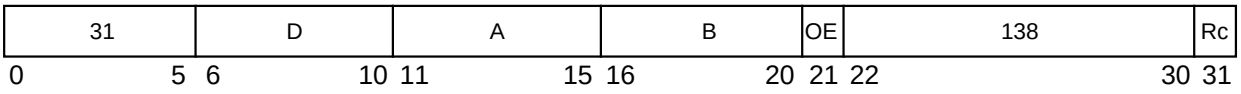
  NOTE: For more information on condition codes see Section 2.1.3, “Condition Register,” and Section 2.1.5, “XER Register,” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

adde*x* adde*x*

Add Extended (x'7C00 0114')

adde **r**D,**r**A,**r**B (OE = 0 Rc = 0)
adde. **r**D,**r**A,**r**B (OE = 0 Rc = 1)
addeo **r**D,**r**A,**r**B (OE = 1 Rc = 0)
addeo. **r**D,**r**A,**r**B (OE = 1 Rc = 1)




```
rD← (rA) + (rB) + XER[CA]
```

The sum (**r**A) + (**r**B) + XER[CA] is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

  NOTE: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).

- XER:

  Affected: CA

  Affected: SO, OV (if OE = 1)

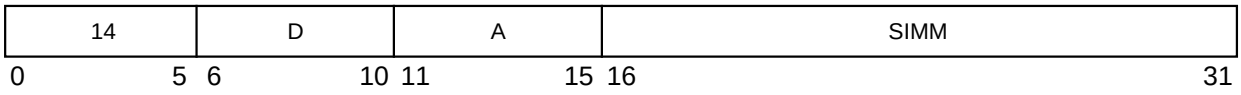
  NOTE: For more information on condition codes see Section 2.1.3, "Condition Register," and Section 2.1.5, "XER Register," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

addi addi

Add Immediate (x'3800 0000')

addi **r**D**,r**A**,**SIMM




```
if rA = 0
        then rD ←EXTS(SIMM)
        else rD←(rA) + EXTS(SIMM)
```

The sum (**r**A|0) + sign extended SIMM is placed into **r**D.

The **addi** instruction is preferred for addition because it sets few status bits. Note that **addi** uses the value 0, not the contents of GPR0, if **r**A = 0.

Other registers altered:

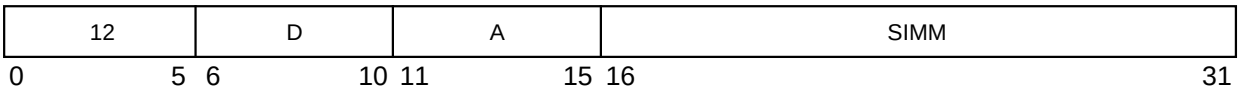
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

addic addic

Add Immediate Carrying (x'3000 0000')

addic **r**D**,r**A**,**SIMM




```
rD← (rA) + EXTS(SIMM)
```

The sum (**r**A) + sign extended SIMM is placed into **r**D.

Other registers altered:

- XER:

  NOTE: Affected: CAFor more information see Section 2.1.5, "XER Register," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Simplified mnemonics:

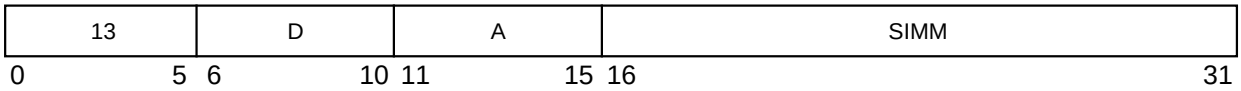
subic rD,rA,value equivalent to addic rD,rA,–value

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

addic. addic.

Add Immediate Carrying and Record (x'3400 0000')

addic. **r**D**,r**A**,**SIMM




```
rD← (rA) + EXTS(SIMM)
```

The sum (**r**A) + the sign extended SIMM is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO

  NOTE: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).

- XER:

  Affected: CA

  NOTE: For more information on condition codes see Section 2.1.3, "Condition Register," and Section 2.1.5, "XER Register," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Simplified mnemonics:

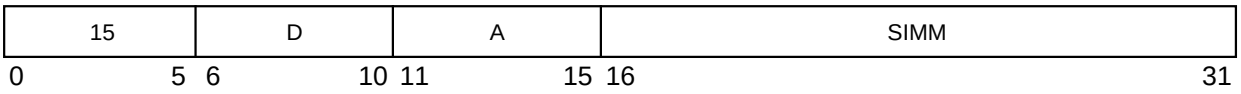
subic.rD**,r**A**,**value equivalent to **addic. r**D**,r**A**,**–value

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

addis addis

Add Immediate Shifted (x'3C00 0000')

addis **r**D,**r**A,SIMM




```
if rA = 0
        then rD← (SIMM || (16)0)
        else rD← (rA) + (SIMM || (16)0)
```

The sum (**r**A|0) + (SIMM || 0x0000) is placed into **r**D.

The **addis** instruction is preferred for addition because it sets few status bits. Note that **addis** uses the value 0, not the contents of GPR0, if **r**A = 0.

Other registers altered:

- None

Simplified mnemonics:

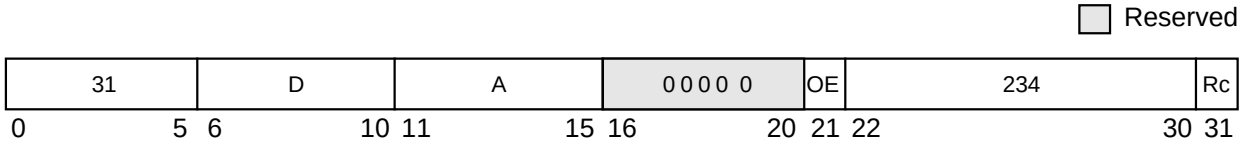
lis	rD,	value equivalent to	addis	rD,0,value
subis	rD,rA,	value equivalent to	addis	rD,rA,–value

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

addme*x* addme*x*

Add to Minus One Extended (x'7C00 01D4')

addme **r**D**,r**A (OE = 0 Rc = 0)
addme. **r**D**,r**A (OE = 0 Rc = 1)
addmeo **r**D**,r**A (OE = 1 Rc = 0)
addmeo. **r**D**,r**A (OE = 1 Rc = 1)




```
rD← (rA) + XER[CA] – 1
```

The sum (**r**A) + XER[CA] + 0xFFFF_FFFF is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

  NOTE: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).

- XER:

  Affected: CA

  Affected: SO, OV (if OE = 1)

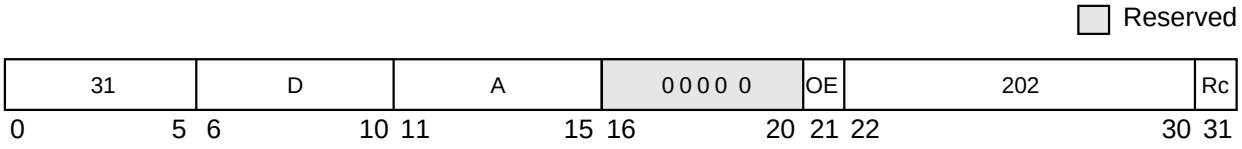
  NOTE: For more information on condition codes see Section 2.1.3, "Condition Register," and Section 2.1.5, "XER Register," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

addze*x* addze*x*

Add to Zero Extended (x'7C00 0194')

addze	**r**D,**r**A	(OE = 0 Rc = 0)
addze.	**r**D,**r**A	(OE = 0 Rc = 1)
addzeo	**r**D,**r**A	(OE = 1 Rc = 0)
addzeo.	**r**D,**r**A	(OE = 1 Rc = 1)




```
rD← (rA) + XER[CA]
```

The sum (**r**A) + XER[CA] is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

  NOTE: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).

- XER:

  Affected: CA

  Affected: SO, OV (if OE = 1)

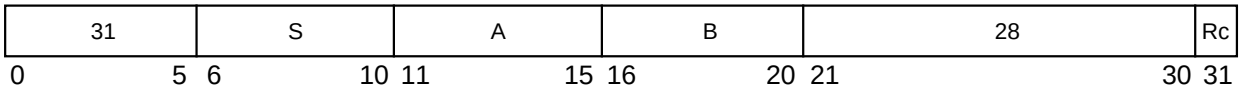
  NOTE: For more information on condition codes see Section 2.1.3, "Condition Register," and Section 2.1.5, "XER Register," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

and*x* and*x*

AND (x’7C00 0038’)

and **r**A**,r**S**,r**B (Rc = 0)
and. **r**A**,r**S**,r**B (Rc = 1)



rA← (**r**S) & (**r**B)

The contents of **r**S are ANDed with the contents of **r**B and the result is placed into **r**A.

Other registers altered:

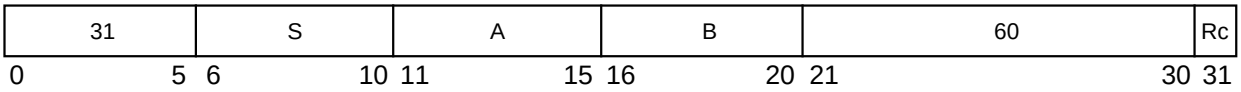
- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

andc*x* andc*x*

AND with Complement (x'7C00 0078')

andc **r**A,**r**S,**r**B (Rc = 0)
andc. **r**A,**r**S,**r**B (Rc = 1)



rA ← (**r**S) & ¬ (**r**B)

The contents of **r**S are ANDed with the one's complement of the contents of **r**B and the result is placed into **r**A.

Other registers altered:

- Condition Register (CR0 field):

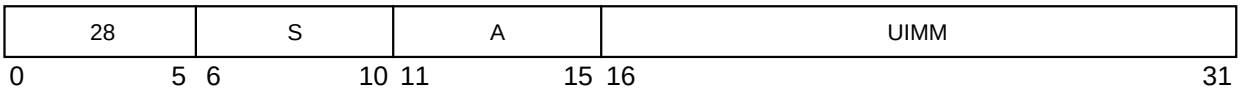
  Affected: LT, GT, EQ, SO (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

andi. andi.

AND Immediate (x’7000 0000’)

andi. **r**A**,r**S**,**UIMM




```
rA← (rS) & ((16)0 || UIMM)
```

The contents of **r**S are ANDed with 0x000 || UIMM and the result is placed into **r**A.

Other registers altered:

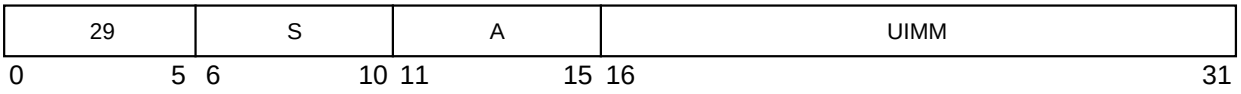
- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

andis. andis.

AND Immediate Shifted (x'7400 0000')

andis. **r**A,**r**S,UIMM



rA← (**r**S) & (UIMM || (16)0)

The contents of **r**S are ANDed with UIMM || 0x0000 and the result is placed into **r**A.

Other registers altered:

- Condition Register (CR0 field):

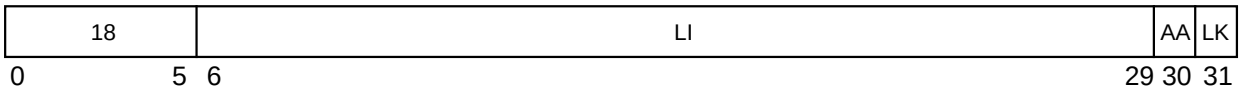
  Affected: LT, GT, EQ, SO

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

b*x* b*x*

Branch (x’4800 0000’)

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




```
if AA = 1
        then NIA ←iea EXTS(LI || 0b00)
        else NIA ←iea CIA + EXTS(LI || 0b00)
if LK = 1
        then LR ←iea CIA + 4
```

target_addr specifies the branch target address.

If AA = 1, then the branch target address is the value LI || 0b00 sign-extended.

If AA = 0, then the branch target address is the sum of LI || 0b00 sign-extended plus the address of this instruction.

If LK = 1, then the effective address of the instruction following the branch instruction is placed into the link register.

Other registers altered:

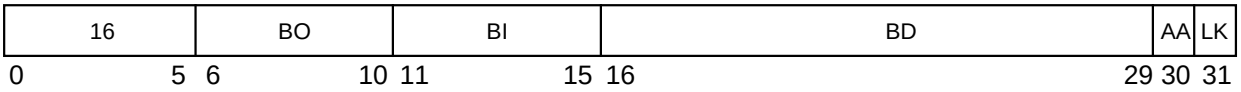
- Link Register (LR) (if LK = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				I

bc*x* bc*x*

Branch Conditional (x'4000 0000')

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




```
if ¬ BO[2] then CTR ← CTR – 1
ctr_ok ← BO[2] | ((CTR ≠ 0) ⊕ BO[3])
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if ctr_ok & cond_ok
    then
         if AA = 1
              then NIA ←iea EXTS(BD || 0b00)
              else NIA ←iea CIA + EXTS(BD || 0b00)
         if LK then LR ←iea CIA + 4
```

target_addr specifies the branch target address.

The BI field specifies the bit in the condition register (CR) to be used as the condition of the branch. The BO field is encoded as described in Table 12-6.

Additional information about BO field encoding is provided in Section 4.2.4.2, "Conditional Branch Control," *in* the *PowerPC Microprocessor Family: The Programming Environments* manual.

NOTE: In this table, *z* indicates a bit that is ignored. The *z* bits should be cleared, as they may be assigned a meaning in some future version of the PowerPC architecture.The *y* bit provides a hint about whether a conditional branch is likely to be taken, and may be used by some PowerPC implementations to improve performance.

Table 12-6. BO Operand Encodings

BO	Description
0000*y*	Decrement the CTR, then branch if the decremented CTR 0 and the condition is FALSE.
0001*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.
001*zy*	Branch if the condition is FALSE.
0100*y*	Decrement the CTR, then branch if the decremented CTR 0 and the condition is TRUE.
0101*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.
011*zy*	Branch if the condition is TRUE.

BO	Description
1*z*00*y*	Decrement the CTR, then branch if the decremented CTR 0.
1*z*01*y*	Decrement the CTR, then branch if the decremented CTR = 0.
1*z*1*zz*	Branch always.

If AA = 0, the branch target address is the sum of BD || 0b00 sign-extended and the address of this instruction.

If AA = 1, the branch target address is the value BD || 0b00 sign-extended.

If LK = 1, the effective address of the instruction following the branch instruction is placed into the link register.

Other registers altered:

Affected: Count Register (CTR) (if BO[2] = 0)

Affected: Link Register (LR) (if LK = 1)

Simplified mnemonics:

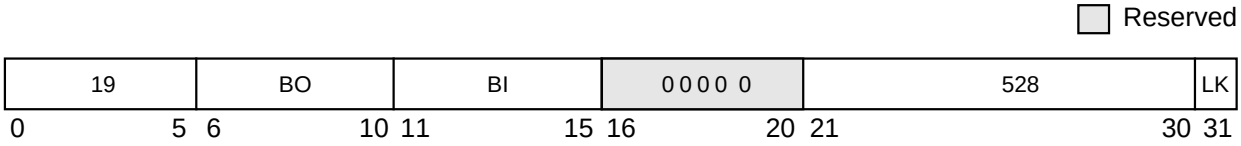
blt target	equivalent to	bc 12,0,target
bne cr2,target	equivalent to	bc 4,10,target
bdnz target	equivalent to	bc 16,0,target

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

bcctr*x* bcctr*x*

Branch Conditional to Count Register (x'4C00 0420')

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




```
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if cond_ok
     then
          NIA ←iea CTR || 0b00
          if LK then LR ←iea CIA + 4
```

The BI field specifies the bit in the condition register to be used as the condition of the branch. The BO field is encoded as described in Table 12-7. Additional information about BO field encoding is provided in Section 4.2.4.2, "Conditional Branch Control," *in* the *PowerPC Microprocessor Family: The Programming Environments* manual.

Table 12-7. BO Operand Encodings

BO	Description
0000*y*	Decrement the CTR, then branch if the decremented CTR 0 and the condition is FALSE.
0001*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.
001*zy*	Branch if the condition is FALSE.
0100*y*	Decrement the CTR, then branch if the decremented CTR 0 and the condition is TRUE.
0101*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.
011*zy*	Branch if the condition is TRUE.
1*z*00*y*	Decrement the CTR, then branch if the decremented CTR 0.
1*z*01*y*	Decrement the CTR, then branch if the decremented CTR = 0.
1*z*1*zz*	Branch always.

In this table, *z* indicates a bit that is ignored.
Note that the *z* bits should be cleared, as they may be assigned a meaning in some future version of the PowerPC architecture.

The *y* bit provides a hint about whether a conditional branch is likely to be taken, and may be used by some PowerPC implementations to improve performance.

The branch target address is CTR[0–29] || 0b00.

If LK = 1, the effective address of the instruction following the branch instruction is placed into the link register.

If the “decrement and test CTR” option is specified (BO[2] = 0), the instruction form is invalid.

Other registers altered:

- Link Register (LR) (if LK = 1)

Simplified mnemonics:

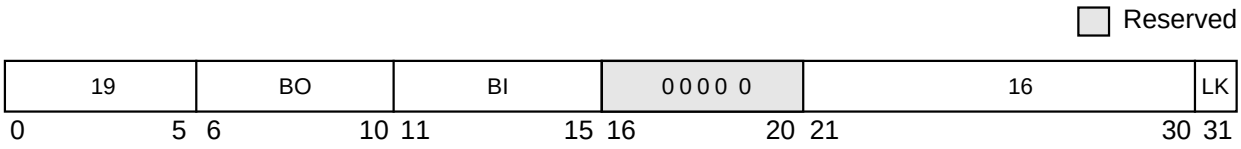
bltctr	equivalent to	bcctr 12,0
bnectrcr2	equivalent to	bcctr 4,10

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XL

bclrx bclrx

Branch Conditional to Link Register (x'4C00 0020')

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




```
if ¬ BO[2] then CTR ← CTR – 1
ctr_ok ← BO[2] | ((CTR ≠   0)⊕ BO[3])
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if ctr_ok & cond_ok
     then
           NIA ←iea LR[0–29] || 0b00
           if LK then LR ←iea CIA + 4
```

The BI field specifies the bit in the condition register to be used as the condition of the branch. The BO field is encoded as described in Table 12-8. Additional information about BO field encoding is provided in Section 4.2.4.2, "Conditional Branch Control," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Table 12-8. BO Operand Encodings

BO	Description
0000*y*	Decrement the CTR, then branch if the decremented CTR 0 and the condition is FALSE.
0001*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.
001*zy*	Branch if the condition is FALSE.
0100*y*	Decrement the CTR, then branch if the decremented CTR 0 and the condition is TRUE.
0101*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.
011*zy*	Branch if the condition is TRUE.
1*z*00*y*	Decrement the CTR, then branch if the decremented CTR 0.
1*z*01*y*	Decrement the CTR, then branch if the decremented CTR = 0.
1*z*1*zz*	Branch always.

If the BO field specifies that the CTR is to be decremented, the entire 32-bit CTR is decremented .

In this table, *z* indicates a bit that is ignored.
Note that the *z* bits should be cleared, as they may be assigned a meaning in some future version of the PowerPC architecture.

The *y* bit provides a hint about whether a conditional branch is likely to be taken, and may be used by some PowerPC implementations to improve performance.

The branch target address is LR[0–29] || 0b00.

If LK = 1, then the effective address of the instruction following the branch instruction is placed into the link register.

Other registers altered:

- Count Register (CTR) (if BO[2] = 0)
- Link Register (LR) (if LK = 1)

Simplified mnemonics:

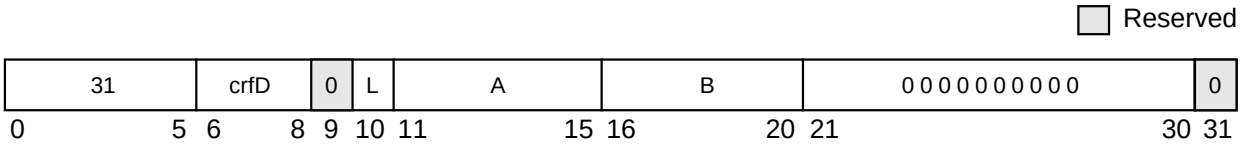
bltlr	equivalent to	bclr 12,0
bnelr cr2	equivalent to	bclr 4,10
bdnzlr	equivalent to	bclr 16,0

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XL

cmp cmp

Compare (x'7C00 0000')

cmp **crf**D,L,**r**A,**r**B




```
a ← (rA)
b ← (rB)
if  a < b
    then c ← 0b100
    else if a > b
            then c ← 0b010
            else c ← 0b001
CR[(4 * crfD)-(4 * crfD + 3)] ← c || XER[SO]
```

The contents of **r**A are compared with the contents of **r**B, treating the operands as signed integers. The result of the comparison is placed into CR field **crf**D.

If L = 1 the instruction form is invalid.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):

  Affected: LT, GT, EQ, SO

Simplified mnemonics:

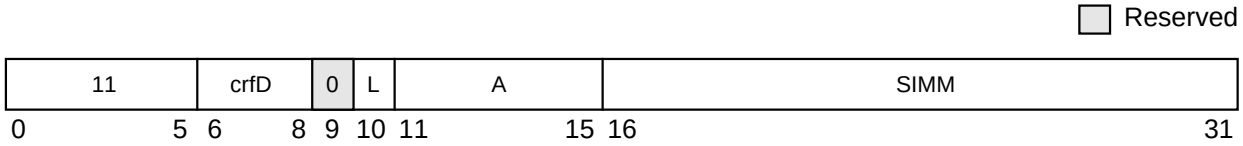
cmpdrA,**r**B	equivalent to	**cmp 0,1,r**A,**r**B
cmpwcr3,rA,**r**B	equivalent to	**cmp 3,0,r**A,**r**B

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

cmpi cmpi

Compare Immediate (x’2C00 0000’)

cmpi **crf**D,L,**r**A,SIMM




```
a ← (rA)
if   a < EXTS(SIMM)
     then c ← 0b100
     else if a > EXTS(SIMM)
               then c ← 0b010
               else c ← 0b001
CR[(4 * crfD)-(4 * crfD + 3)] ← c || XER[SO]
```

The contents of **r**A are compared with the sign-extended value of the SIMM field, treating the operands as signed integers. The result of the comparison is placed into CR field **crf**D.

f L = 1 the instruction form is invalid.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):

  Affected: LT, GT, EQ, SO

Simplified mnemonics:

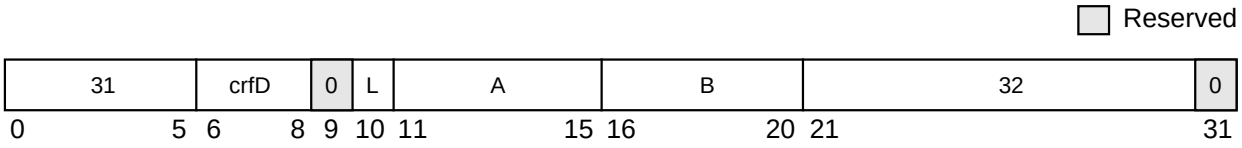
cmpdirA,value	equivalent to	**cmpi 0,1,r**A,value
cmpwi cr3,rA,value	equivalent to	**cmpi 3,0,r**A,value

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

cmpl cmpl

Compare Logical (x'7C00 0040')

cmpl **crf**D,L,**r**A,**r**B




```
a ← (rA)
b ← (rB)
if   a <U b
     then c ← 0b100
     else if a >U b
               then c ← 0b010
               else c ← 0b001
CR[(4 * crfD)-(4 * crfD + 3)] ← c || XER[SO]
```

The contents of **r**A are compared with the contents of **r**B, treating the operands as unsigned integers. The result of the comparison is placed into CR field **crf**D.

If L = 1 the instruction form is invalid.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):

  Affected: LT, GT, EQ, SO

Simplified mnemonics:

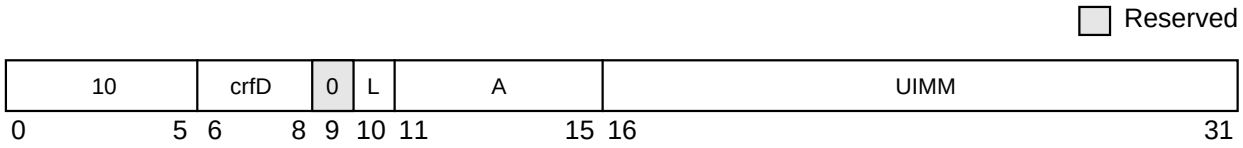
cmpldrA,**r**B	equivalent to	**cmpl 0,1,r**A,**r**B
cmplw cr3,rA,**r**B	equivalent to	**cmpl 3,0,r**A,**r**B

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

cmpli cmpli

Compare Logical Immediate (x'2800 0000')

cmpli **crf**D**,**L**,r**A**,**UIMM




```
a ← (rA)
if   a <U ((16)0 || UIMM)
     then c ← 0b100
     else if a >U ((16)0 || UIMM)
               then c ← 0b010
               else c ← 0b001
CR[(4 * crfD)-(4 * crfD + 3)] ← c || XER[SO]
```

The contents of **r**A are compared with 0x0000 || UIMM, treating the operands as unsigned integers. The result of the comparison is placed into CR field **crf**D.

If L = 1 the instruction form is invalid.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):

  Affected: LT, GT, EQ, SO

Simplified mnemonics:

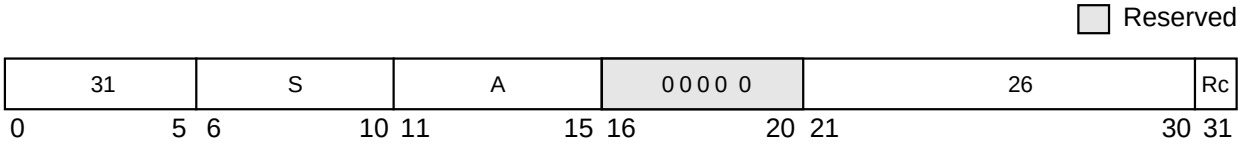
cmpldir A**,**value	equivalent to	**cmpli 0,1,r**A**,**value
cmplwi cr3,rA**,**value	equivalent to	**cmpli 3,0,r**A**,**value

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

cntlzw*x* cntlzw*x*

Count Leading Zeros Word (x'7C00 0034')

cntlzw **r**A,**r**S (Rc = 0)
cntlzw. **r**A,**r**S (Rc = 1)




```
n ← 0
do while n < 32
     if rS[n] = 1 then leave
     n ← n + 1
rA ← n
```

A count of the number of consecutive zero bits starting at bit 0of **r**S is placed into **r**A. This number ranges from 0 to 32, inclusive.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

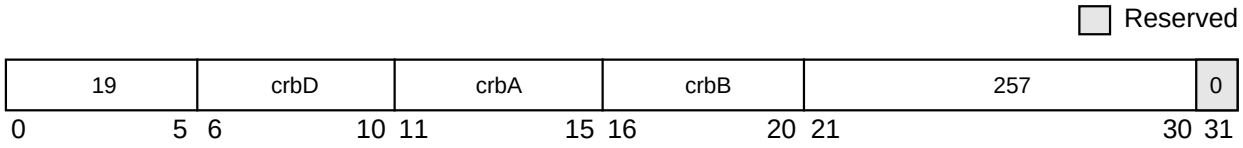
NOTE: If Rc = 1, then LT is cleared in the CR0 field.

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

crand crand

Condition Register AND (x'4C00 0202')

crand **crb**D,**crb**A,**crb**B



CR[**crb**D] ← CR[**crb**A] & CR[**crb**B]

The bit in the condition register specified by **crb**A is ANDed with the bit in the condition register specified by **crb**B. The result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition Register:

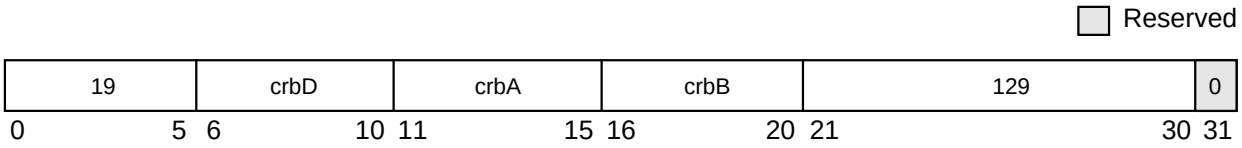
  Affected: Bit specified by operand crbD

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XL

crandc crandc

Condition Register AND with Complement (x'4C00 0102')

crandc **crb**D**,crb**A**,crb**B



CR[**crb**D] ← CR[**crb**A] & ¬ CR[**crb**B]

The bit in the condition register specified by **crb**A is ANDed with the complement of the bit in the condition register specified by **crb**B and the result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition Register:

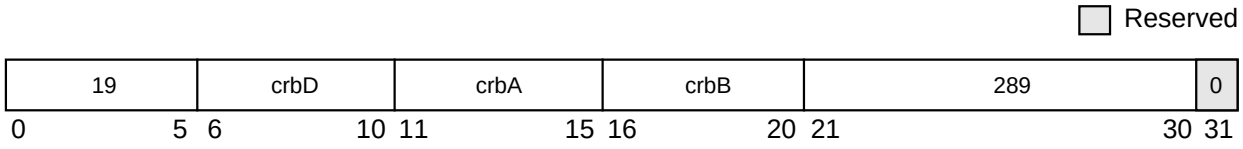
  Affected: Bit specified by operand crbD

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XL

creqv creqv

Condition Register Equivalent (x'4C00 0242')

creqv **crb**D**,crb**A**,crb**B



CR[**crb**D] ← CR[**crb**A] ≡ CR[**crb**B]

The bit in the condition register specified by **crb**A is XORed with the bit in the condition register specified by **crb**B and the complemented result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition Register:

  Affected: Bit specified by operand **crb**D

Simplified mnemonics:

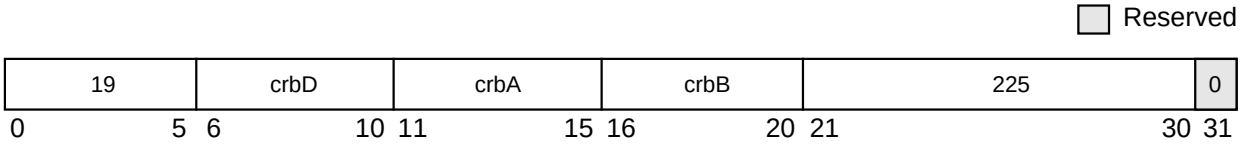
crse crbD equivalent to **creqv crb**D**,crb**D**,crb**D

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XL

crnand crnand

Condition Register NAND (x'4C00 01C2')

crnand **crb**D**,crb**A**,crb**B



CR[**crb**D] ← ¬ (CR[**crb**A] & CR[**crb**B])

The bit in the condition register specified by **crb**A is ANDed with the bit in the condition register specified by **crb**B and the complemented result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition Register:

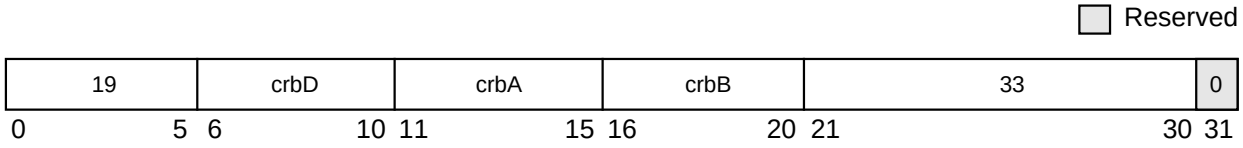
  Affected: Bit specified by operand **crb**D

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XL

crnor crnor

Condition Register NOR (x'4C00 0042')

crnor **crb**D,**crb**A,**crb**B



CR[**crb**D] ← ¬ (CR[**crb**A] | CR[**crb**B])

The bit in the condition register specified by **crb**A is ORed with the bit in the condition register specified by **crb**B and the complemented result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition Register:

  Affected: Bit specified by operand **crb**D

Simplified mnemonics:

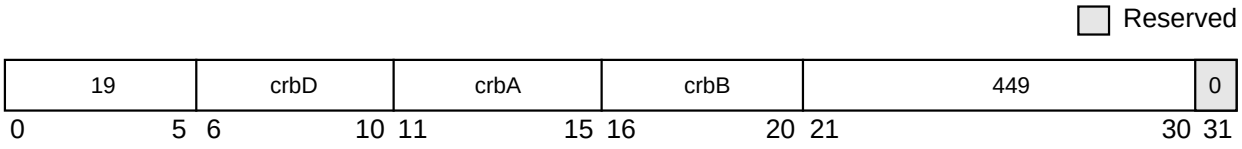
crnot crbD,crbA equivalent to crnor crbD,crbA,crbA

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XL

cror cror

Condition Register OR (x'4C00 0382')

cror **crb**D,**crb**A,**crb**B



CR[**crb**D] ← CR[**crb**A] | CR[**crb**B]

The bit in the condition register specified by **crb**A is ORed with the bit in the condition register specified by **crb**B. The result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition Register:

  Affected: Bit specified by operand **crb**D

Simplified mnemonics:

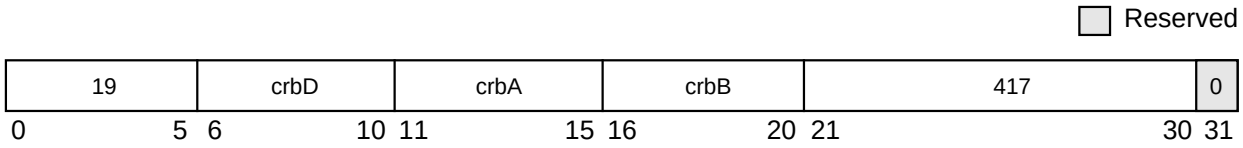
crmove crbD,crbA equivalent to cror crbD,crbA,crbA

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XL

crorc crorc

Condition Register OR with Complement (x'4C00 0342')

crorc **crb**D,**crb**A,**crb**B



CR[**crb**D] ← CR[**crb**A] | ¬ CR[**crb**B]

The bit in the condition register specified by **crb**A is ORed with the complement of the condition register bit specified by **crb**B and the result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition Register:

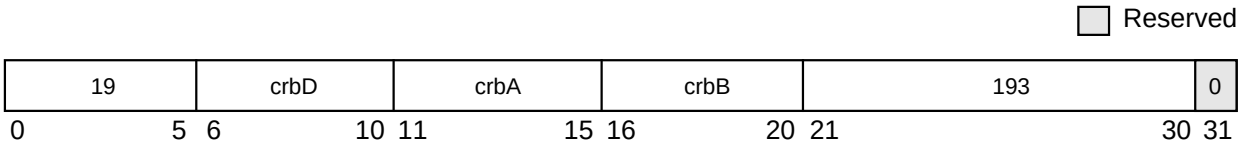
  Affected: Bit specified by operand **crb**D

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XL

crxor crxor

Condition Register XOR (x'4C00 0182')

crxor **crb**D**,crb**A**,crb**B



CR[**crb**D] ← CR[**crb**A] ⊕ CR[**crb**B]

The bit in the condition register specified by **crb**A is XORed with the bit in the condition register specified by **crb**B and the result is placed into the condition register specified by **crb**D.

Other registers altered:

- Condition Register:

  Affected: Bit specified by **crb**D

Simplified mnemonics:

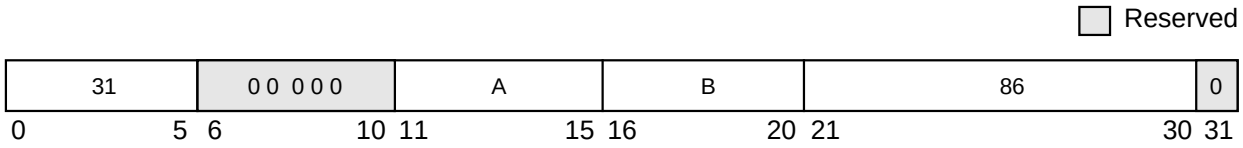
crclr crbD equivalent to **crxor crb**D**,crb**D**,crb**D

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XL

dcbf dcbf

Data Cache Block Flush (x'7C00 00AC')

dcbf **r**A**,r**B



EA is the sum (**r**A|0) + (**r**B).

The **dcbf** instruction invalidates the block in the data cache addressed by EA, copying the block to memory first, if there is any dirty data in it. Unmodified block—Invalidates the block in the processor's data cache. The list below describes the action taken if the block containing the byte addressed by EA is or is not in the cacche:

— Unmodified block—Invalidates the block in the processor's data cache.

— Modified block—Copies the block to memory. Invalidates the block in the processor's data cache.

— Absent block (target block not in cache)—No action is taken.

The function of this instruction is independent of the write-through, write-back and caching-inhibited/allowed modes of the block containing the byte addressed by EA. This instruction is treated as a load from the addressed byte with respect to address translation and memory protection. It is also treated as a load for referenced and changed bit recording except that referenced and changed bit recording may not occur.

When HID2[LCE] = 1 and the byte addressed by EA is in the locked cache, the instruction is not forwarded to the L2 cache for sector invalidation/push, nor forwarded to the 60x bus for broadcast. Otherwise, the instruction will be forwarded to the L2 cache and to the 60x bus as described in Sections 3.4.2.4 and 9.2.1, in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Other registers altered:

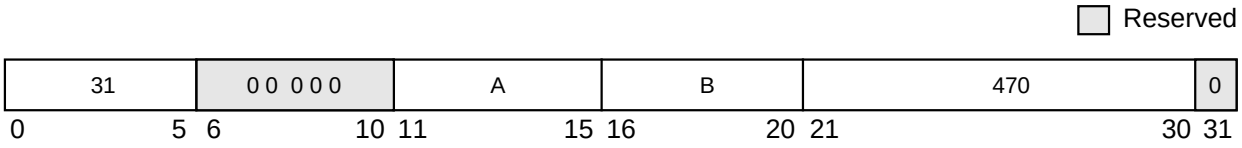
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
VEA				X

dcbi dcbi

Data Cache Block Invalidate (x'7C00 03AC')

dcbi **r**A,**r**B



EA is the sum (**r**A|0) + (**r**B).

The action taken is dependent on the memory mode associated with the block containing the byte addressed by EA and on the state of that block. The list below describes the action taken if the block containing the byte addressed by EA is or is not in the cache.

— Unmodified block—Invalidates the block in the processor's data cache.
— Modified block—Invalidates the block in the processor's data cache. (Discards the modified contents.)
— Absent block (target block not in cache)—No action is taken.

When data address translation is enabled, MSR[DR] = 1, and the virtual address has no translation, a DSI exception occurs.

The function of this instruction is independent of the write-through and caching-inhibited/allowed modes of the block containing the byte addressed by EA. This instruction operates as a store to the addressed byte with respect to address translation and protection. The referenced and changed bits are modified appropriately.

When HID2[LCE] = 1 and the byte addressed by EA is in the locked cache, the instruction is not forwarded to the L2 cache for sector invalidation, nor forwarded to the 60x bus for broadcast. Otherwise, the instruction will be forwarded to teh L2 cache and to the 60x bus as described in Sections 3.4.2.4 and 9.2.1, in the *PowerPC Microprocessor Family: The Programming Environments* manual.

This is a supervisor-level instruction.

Other registers altered:

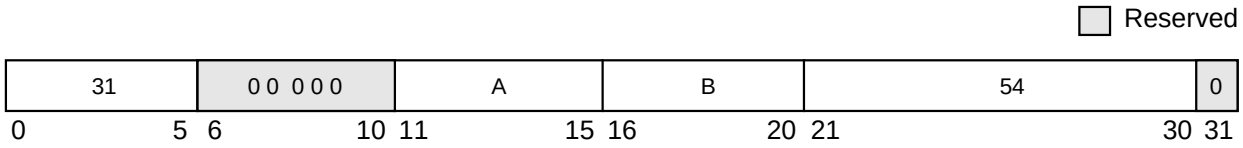
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
VEA	Yes			X

dcbst dcbst

Data Cache Block Store (x'7C00 006C')

dcbst **r**A**,r**B



EA is the sum (**r**A|0) + (**r**B).

The **dcbst** instruction executes as follows:

- If the block containing the byte addressed by EA is in coherency-not-required mode, and a block containing the byte addressed by EA is in the data cache of this processor and has been modified, the writing of it to main memory is initiated.

The function of this instruction is independent of the write-through and caching-inhibited/allowed modes of the block containing the byte addressed by EA.

The processor treats this instruction as a load from the addressed byte with respect to address translation and memory protection. It is also treated as a load for referenced and changed bit recording except that referenced and changed bit recording may not occur.

When HID2[LCE] = 1 and the byte addressed by EA is in the locked cache, the instruction is not forwarded to the L2 cache for sector invalidation/push, nor forwarded to the 60x bus for broadcast. Otherwise, the instruction will be forwarded to the L2 cache and to the 60x bus as described in Sections 3.4.2.4 and 9.2.1, in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Other registers altered:

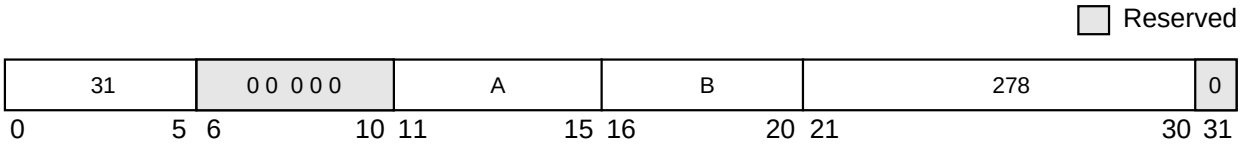
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
VEA				X

dcbt dcbt

Data Cache Block Touch (x'7C00 022C')

dcbt **r**A,**r**B



EA is the sum (**r**A|0) + (**r**B).

This instruction is a hint that performance will possibly be improved if the block containing the byte addressed by EA is fetched into the data cache, because the program will probably soon load from the addressed byte. If the block is caching-inhibited, the hint is ignored and the instruction is treated as a no-op. Executing **dcbt** does not cause the system alignment error handler to be invoked.

If HID2[LCE] = 1 and the byte addressed by EA is in neither the locked nor the normal cache, then this instruction loads the cache line into the "normal" cache.

This instruction is treated as a load from the addressed byte with respect to address translation, memory protection, and reference and change recording except that referenced and changed bit recording may not occur. Additionally, no exception occurs in the case of a translation fault or protection violation.

The program uses the **dcbt** instruction to request a cache block fetch before it is actually needed by the program. The program can later execute load instructions to put data into registers. However, the processor is not obliged to load the addressed block into the data cache. Note that this instruction is defined architecturally to perform the same functions as the **dcbtst** instruction. Both are defined in order to allow implementations to differentiate the bus actions when fetching into the cache for the case of a load and for a store.

Other registers altered:

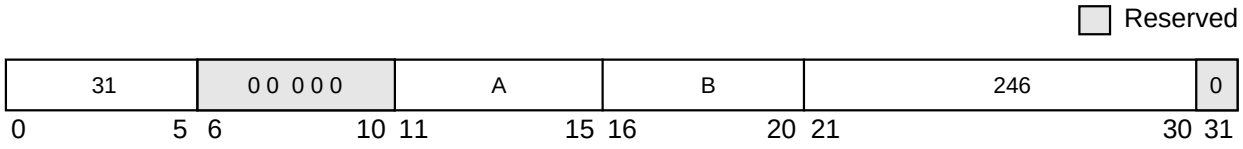
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
VEA				X

dcbtst dcbtst

Data Cache Block Touch for Store (x'7C00 01EC')

dcbtst **r**A,**r**B



EA is the sum (**r**A|0) + (**r**B).

This instruction is a hint that performance will possibly be improved if the block containing the byte addressed by EA is fetched into the data cache, because the program will probably soon store from the addressed byte. If the block is caching-inhibited, the hint is ignored and the instruction is treated as a no-op. Executing **dcbtst** does not cause the system alignment error handler to be invoked.

If HID2[LCE] = 1 and the byte addressed by EA is in neither the locked nor the normal cache, then this instruction loads the cache line into the "normal" cache.

This instruction is treated as a load from the addressed byte with respect to address translation, memory protection, and reference and change recording except that referenced and changed bit recording may not occur. Additionally, no exception occurs in the case of a translation fault or protection violation.

The program uses **dcbtst** to request a cache block fetch to potentially improve performance for a subsequent store to that EA, as that store would then be to a cached location. However, the processor is not obliged to load the addressed block into the data cache. Note that this instruction is defined architecturally to perform the same functions as the **dcbt** instruction. Both are defined in order to allow implementations to differentiate the bus actions when fetching into the cache for the case of a load and for a store.

Other registers altered:

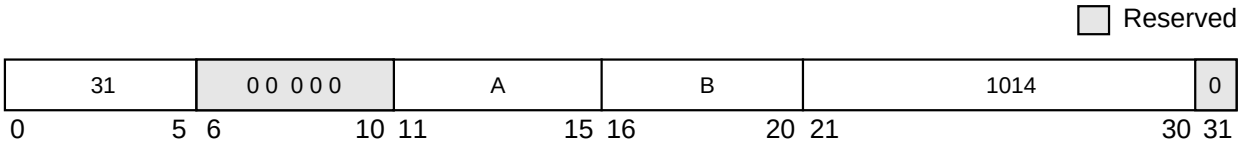
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
VEA				X

dcbz dcbz

Data Cache Block Clear to Zero (x'7C00 07EC')

dcbz **r**A,**r**B



EA is the sum (**r**A|0) + (**r**B).

This instruction is treated as a store to the addressed byte with respect to address translation, memory protection, referenced and changed recording. It is also treated as a store with respect to the ordering enforced by **eieio** and the ordering enforced by the combination of caching-inhibited and guarded attributes for a page (or block).

The **dcbz** instruction executes as follows:

- If the cache block containing the byte addressed by EA is in the data cache, all bytes are cleared and the cache line is matked "M"..
- If the cache block containing the byte addressed by EA is not in the data cache and the corresponding memory page or block is caching-allowed, the cache block is allocated (and made valid) in the data cache (or in the normal cache if HID2[LCE] = 1) without fetching the block from main memory, and all bytes are cleared.
- If the page containing the byte addressed by EA is in caching-inhibited or write-through mode, either all bytes of main memory that correspond to the addressed cache block are cleared or the alignment exception handler is invoked. The exception handler can then clear all bytes in main memory that correspond to the addressed cache block.
- If the cache block containing the byte addressed by EA is in coherency-required mode, and the cache block exists in the data cache(s) of any other processor(s), it is kept coherent in those caches (i.e. the processor performs the appropriate bus transactions to enforce this).

Other registers altered:

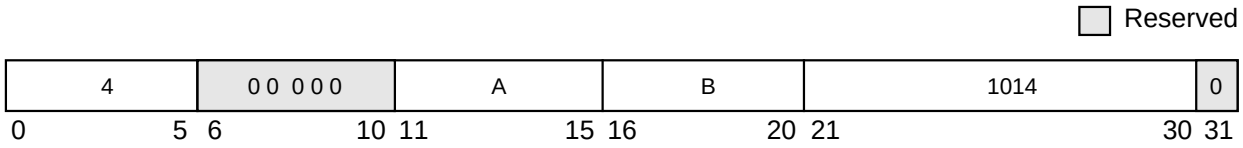
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
VEA				X

dcbz_l dcbz_l

Data Cache Block Set to Zero Locked (x'1000 07EC')

dcbz_l **r**A,**r**B



EA is the sum (**r**A|0) + (**r**B).

If HID2[LCE] = 0 then the invalid instruction error handler is envolked.

When HID2[LCE] = 1, the **dcbz**_l instruction executes as follows:

- If the cache block containing the byte addressed by EA is neither in the "locked" nor in the "normal" data cache, the block is allocated in the "locked" data cache without fetching the block from main memory. All bytes are cleared and the block is marked as M (modified). Cache block allocation is done using the psudo-LRU used rule among the four ways in the locked cache.
- If the cache block containing the byte addressed by EA is already either in the "locked" or in the "normal" data cache, all bytes are cleared and the block is marked M (modified). The hardware indicates this situation by setting HID2[DCHERR] to 1 and raising a Machine Check condition as described in Section 9.2.2.2.1, in the *PowerPC Microprocessor Family: The Programming Environments* manual.
- The dcbz_l instruction is not forwarded to the L2 cache nor the 60x bus for broadcast.
  NOTE: The data cache should be invalidated prior to setting HID2[LCE]=1.

Other registers altered:

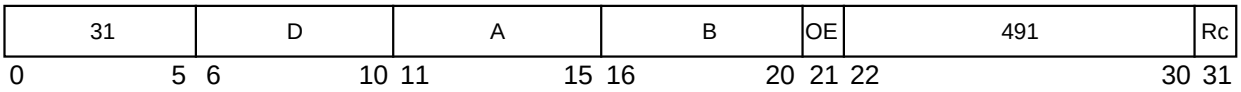
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
VEA		Yes		X

divw*x* divw*x*

Divide Word (x'7C00 03D6')

divw	**r**D,**r**A,**r**B	(OE = 0 Rc = 0)
divw.	**r**D,**r**A,**r**B	(OE = 0 Rc = 1)
divwo	**r**D,**r**A,**r**B	(OE = 1 Rc = 0)
divwo.	**r**D,**r**A,**r**B	(OE = 1 Rc = 1)




```
dividend ← (rA)
divisor ← rB)
rD ← dividend  / divisor
```

The dividend is the contents of **r**A. The divisor is the contents of **r**B. The remainder is not supplied as a result. Both the operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies the equation—dividend = (quotient * divisor) + r where 0 r < |divisor| (if the dividend is non-negative), and –|divisor| < r 0 (if the dividend is negative).

If an attempt is made to perform either of the divisions—0x8000_0000 –1 or <anything> 0, then the contents of **r**D are undefined, as are the contents of the LT, GT, and EQ bits of the CR0 field (if Rc = 1). In this case, if OE = 1 then OV is set.

The 32-bit signed remainder of dividing the contents of **r**A by the contents of **r**B can be computed as follows, except in the case that the contents of **r**A = -2^{31} and the contents of **r**B = –1.

divw	**r**D,**r**A,**r**B# **r**D = quotient
mullw	**r**D,**r**D,**r**B# **r**D = quotient ∗ divisor
subf	**r**D,**r**D,**r**A# **r**D = remainder

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

- XER:

  Affected: SO, OV (if OE = 1)

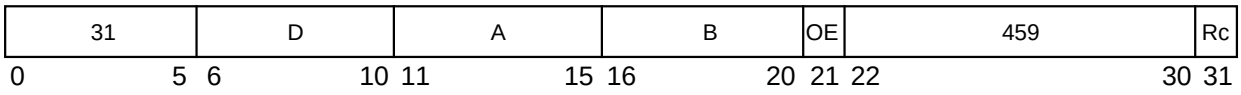
  NOTE: For more information on condition codes see Section 2.1.3, "Condition Register," and Section 2.1.5, "XER Register," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

divwu*x* divwu*x*

Divide Word Unsigned (x'7C00 0396')

divwu **r**D,**r**A,**r**B (OE = 0 Rc = 0)
divwu. **r**D,**r**A,**r**B (OE = 0 Rc = 1)
divwuo **r**D,**r**A,**r**B (OE = 1 Rc = 0)
divwuo. **r**D,**r**A,**r**B (OE = 1 Rc = 1)




```
dividend ← (rA)
divisor ← (rB)
rD← dividend   divisor
```

The dividend is the contents of **r**A. The divisor is the contents of **r**B. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc = 1 the first three bits of CR0 field are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies the equation—dividend = (quotient ∗ divisor) + r (where 0 r < divisor). If an attempt is made to perform the division—<anything> 0—then the contents of **r**D are undefined as are the contents of the LT, GT, and EQ bits of the CR0 field (if Rc = 1). In this case, if OE = 1 then OV is set.

The 32-bit unsigned remainder of dividing the contents of **r**A by the contents of **r**B can be computed as follows:

```
divwurD,rA,rB      # rD = quotient
mullw rD,rD,rB     # rD = quotient ∗ divisor
subf  rD,rD,rA      # rD = remainder
```

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

- XER:

  Affected: SO, OV (if OE = 1)

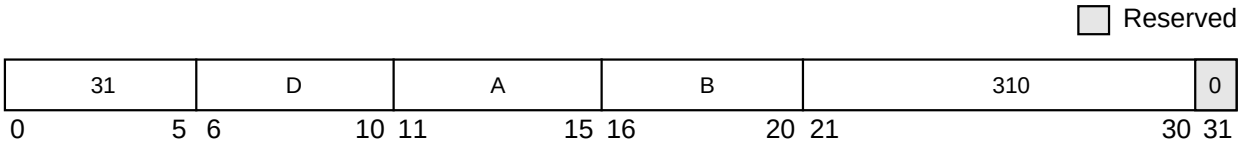
  NOTE: For more information on condition codes see Section 2.1.3, "Condition Register," and Section 2.1.5, "XER Register," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

eciwx eciwx

External Control In Word Indexed (x'7C00 026C')

eciwx **r**D,**r**A,**r**B



The **eciwx** instruction and the EAR register can be very efficient when mapping special devices such as graphics devices that use addresses as pointers.

```
if rA = 0
     then b ← 0
     else b← (rA)
EA ← b + (rB)
paddr ← address translation of EA
send load word request for paddr to device identified by EAR[RID]
rD ← word from device
```

EA is the sum (**r**A|0) + (**r**B).

A load word request for the physical address (referred to as real address in the architecture specification) corresponding to EA is sent to the device identified by EAR[RID], bypassing the cache. The word returned by the device is placed in **r**D.

EAR[E] must be 1. If it is not, a DSI exception is generated.

EA must be a multiple of four. If it is not, one of the following occurs:

- A system alignment exception is generated.
- A DSI exception is generated (possible only if EAR[E] = 0).
- The results are boundedly undefined.

The **eciwx** instruction is supported for EAs that reference memory segments in which SR[T] = 1(or STE[T] = 1) and for EAs mapped by the DBAT registers. If the EA references a direct-store segment (SR[T] = 1 or STE[T] = 1), either a DSI exception occurs or the results are boundedly undefined. However, note that the direct-store facility is being phased out of the architecture and will not likely be supported in future devices. Thus, software should not depend on its effects.

If this instruction is executed when MSR[DR] = 0 (real addressing mode), the results are boundedly undefined.

This instruction is treated as a load from the addressed byte with respect to address translation, memory protection, referenced and changed bit recording, and the ordering performed by **eieio**.

This instruction is optional in the PowerPC architecture.

Other registers altered:

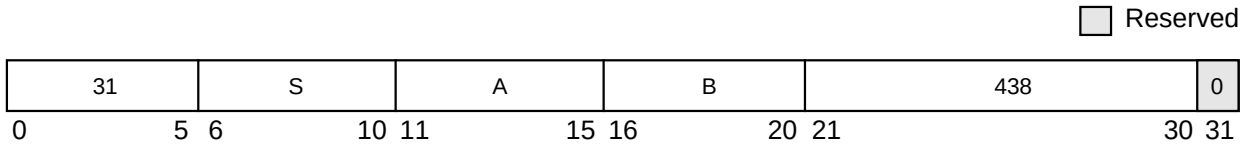
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
VEA			X	X

ecowx ecowx

External Control Out Word Indexed (x'7C00 036C')

ecowx **r**S,**r**A,**r**B



The **ecowx** instruction and the EAR register can be very efficient when mapping special devices such as graphics devices that use addresses as pointers.

```
if rA = 0
    then b ← 0
    else b ← (rA)
EA ← b + (rB)
paddr ← address translation of EA
send store word request for paddr to device identified by EAR[RID]
send rS to device
```

EA is the sum (**r**A|0) + (**r**B).

A store word request for the physical address corresponding to EA and the contents of **r**S are sent to the device identified by EAR[RID], bypassing the cache.

EAR[E] must be 1, if it is not, a DSI exception is generated.

EA must be a multiple of four. If it is not, one of the following occurs:

- A system alignment exception is generated.
- A DSI exception is generated (possible only if EAR[E] = 0).
- The results are boundedly undefined.

The **ecowx** instruction is supported for effective addresses that reference memory segments in which SR[T] = 0 or STE[T] = 0), and for EAs mapped by the DBAT registers. If the EA references a direct-store segment (SR[T] = 1 or STE[T] = 1), either a DSI exception occurs or the results are boundedly undefined. However, note that the direct-store facility is being phased out of the architecture and will not likely be supported in future devices. Thus, software should not depend on its effects.

If this instruction is executed when MSR[DR] = 0 (real addressing mode), the results are boundedly undefined.

This instruction is treated as a store from the addressed byte with respect to address translation, memory protection, and referenced and changed bit recording, and the ordering performed by **eieio**. Note that software synchronization is required in order to ensure that the data access is performed in program order with respect to data accesses caused by other store or **ecowx** instructions, even though the addressed byte is assumed to be caching-inhibited and guarded.

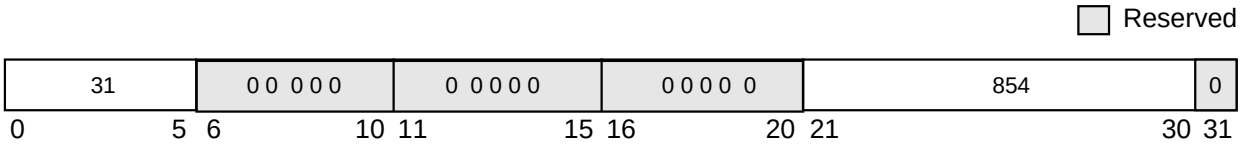
This instruction is optional in the PowerPC architecture.

Other registers altered: None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
VEA			X	X

eieio eieio

Enforce In-Order Execution of I/O (x'7C00 06AC')



The **eieio** instruction provides an ordering function for the effects of load and store instructions executed by a processor. These loads and stores are divided into two sets, which are ordered separately. The memory accesses caused by a **dcbz** or a **dcba** instruction are ordered like a store. The two sets follow:

1. Loads and stores to memory that is both caching-inhibited and guarded, and stores to memory that is write-through required.

   The **eieio** instruction controls the order in which the accesses are performed in main memory. It ensures that all applicable memory accesses caused by instructions preceding the **eieio** instruction have completed with respect to main memory before any applicable memory accesses caused by instructions following the **eieio** instruction access main memory. It acts like a barrier that flows through the memory queues and to main memory, preventing the reordering of memory accesses across the barrier. No ordering is performed for **dcbz** if the instruction causes the system alignment error handler to be invoked.

   All accesses in this set are ordered as a single set—that is, there is not one order for loads and stores to caching-inhibited and guarded memory and another order for stores to write-through required memory.

2. Stores to memory that have all of the following attributes—caching-allowed, write-through not required, and memory-coherency required.

   The **eieio** instruction controls the order in which the accesses are performed with respect to coherent memory. It ensures that all applicable stores caused by instructions preceding the **eieio** instruction have completed with respect to coherent memory before any applicable stores caused by instructions following the **eieio** instruction complete with respect to coherent memory.

With the exception of **dcbz** and **dcba**, **eieio** does not affect the order of cache operations (whether caused explicitly by execution of a cache management instruction, or implicitly by the cache coherency mechanism). For more information, refer to Chapter 5, "Cache Model and Memory Coherency" of the *PowerPC Microprocessor Family: The Programming Environments* manual. The **eieio** instruction does not affect the order of accesses in one set with respect to accesses in the other set.

The **eieio** instruction may complete before memory accesses caused by instructions preceding the **eieio** instruction have been performed with respect to main memory or coherent memory as appropriate.

The **eieio** instruction is intended for use in managing shared data structures, in accessing memory-mapped I/O, and in preventing load/store combining operations in main memory. For the first use, the shared data structure and the lock that protects it must be altered only by stores that are in the same set (1 or 2; see previous discussion). For the second use, **eieio** can be thought of as placing a barrier into the stream of memory accesses issued by a processor, such that any given memory access appears to be on the same side of the barrier to both the processor and the I/O device.

Because the processor performs store operations in order to memory that is designated as both caching-inhibited and guarded (refer to Section 5.1.1, “Memory Access Ordering” in the *PowerPC Microprocessor Family: The Programming Environments* manual), the **eieio** instruction is needed for such memory only when loads must be ordered with respect to stores or with respect to other loads.

Note that the **eieio** instruction does not connect hardware considerations to it such as multiprocessor implementations that send an **eieio** address-only broadcast (useful in some designs). For example, if a design has an external buffer that re-orders loads and stores for better bus efficiency, the **eieio** broadcast signals to that buffer that previous loads/stores (marked caching-inhibited, guarded, or write-through required) must complete before any following loads/stores (marked caching-inhibited, guarded, or write-through required).

Other registers altered:

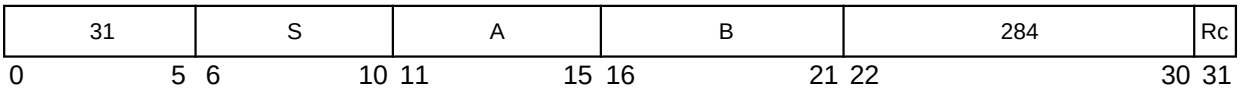
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
VEA				X

eqv*x* eqv*x*

Equivalent (x’7C00 0238’)

eqv **r**A,**r**S,**r**B (Rc = 0)
eqv. **r**A,**r**S,**r**B (Rc = 1)



rA ← (**r**S) ≡ (**r**B)

The contents of **r**S are XORed with the contents of **r**B and the complemented result is placed into **r**A.

Other registers altered:

- Condition Register (CR0 field):

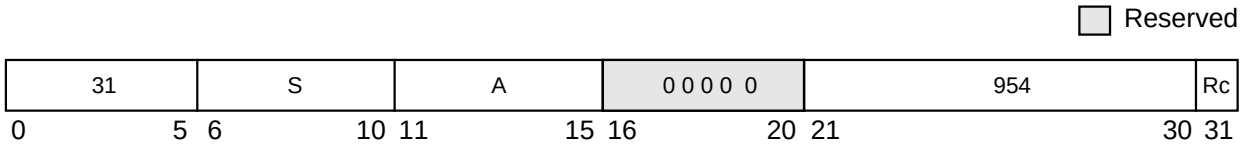
  Affected: LT, GT, EQ, SO (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

extsb*x* extsb*x*

Extend Sign Byte (x'7C00 0774')

extsb **r**A,**r**S (Rc = 0)
extsb. **r**A,**r**S (Rc = 1)




```
S ← rS[24]
rA[24-31] ← rS[24-31]
rA[0–23] ← (24)S
```

The contents of the low-order eight bits of **r**S are placed into the low-order eight bits of **r**A.

Bit 24 of **r**S is placed into the remaining bits of **r**A.

Other registers altered:

- Condition Register (CR0 field):

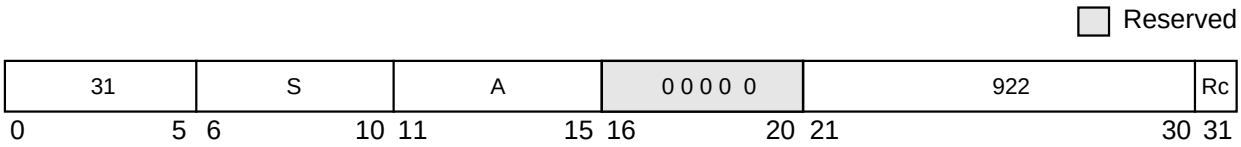
  Affected: LT, GT, EQ, SO (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

extsh*x* extsh*x*

Extend Sign Half Word (x'7C00 0734')

extsh **r**A,**r**S (Rc = 0)
extsh. **r**A,**r**S (Rc = 1)




```
S ← rS[16]
rA[16-31]← rS[16-31]
rA[0–15] ← (16)S
```

The contents of the low-order 16 bits of **r**S are placed into the low-order 16 bits of **r**A[16-31]. Bit 48 of **r**S is placed into the remaining bits of **r**A.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

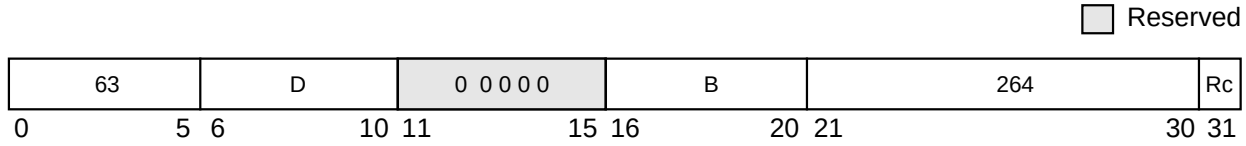
PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

fabsx

fabsx

Floating Absolute Value (x'FC00 0210')

fabs **fr**D,**fr**B (Rc = 0)
fabs. **fr**D,**fr**B (Rc = 1)



The contents of **fr**B with bit 0 cleared are placed into **fr**D.

Note that the **fabs** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN may be altered by **fabs**. This instruction does not alter the FPSCR.

Other registers altered:

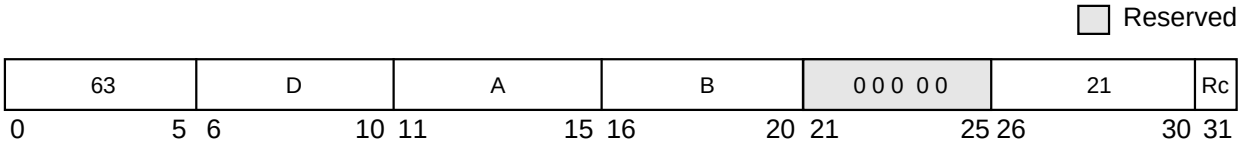
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

fadd*x* fadd*x*

Floating Add (Double-Precision) (x'FC00 002A')

fadd **fr**D**,fr**A**,fr**B (Rc = 0)
fadd. **fr**D**,fr**A**,fr**B (Rc = 1)



The floating-point operand in **fr**A is added to the floating-point operand in **fr**B. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands. All 53 bits in the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered:

- Condition Register (CR1 field):

  Affected: FX, FEX, VX, OX (if Rc = 1)

- Floating-Point Status and Control Register:

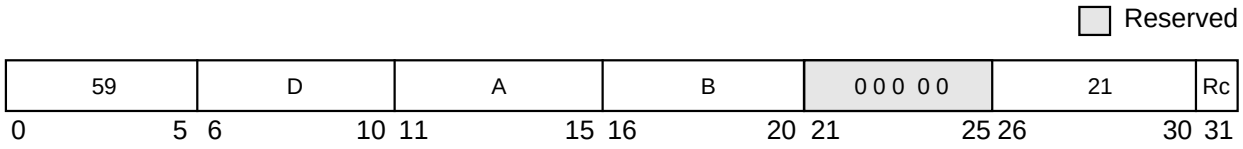
  Affected: FPRF, FR, FI, FX, OX, UX, XX,VXSNAN, VXISI

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

fadds*x* fadds*x*

Floating Add Single (x'EC00 002A')

fadds **fr**D**,fr**A**,fr**B (Rc = 0)
fadds. **fr**D**,fr**A**,fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0
        then frD ← frA + frB
        else frD(ps0) ← frA(ps0) + frB(ps0)
             frD(ps1) ← frD(ps0)
```

The floating-point operand in **fr**A is added to the floating-point operand in **fr**B. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands. All 53 bits in the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

If the HID2[PSE] = 1 then the sum is placed in both **fr**D(ps0) and **fr**D(ps1).

Other registers altered:

- Condition Register (CR1 field):

  Affected: FX, FEX, VX, OX (if Rc = 1)

- Floating-Point Status and Control Register:

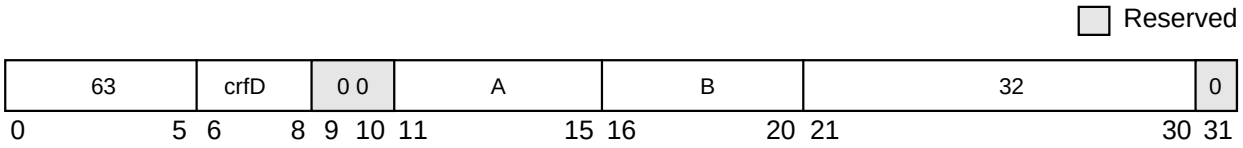
  Affected: FPRF, FR, FI, FX, OX, UX, XX,VXSNAN, VXIS

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

fcmpo fcmpo

Floating Compare Ordered (x'FC00 0040')

fcmpo **crf**D**,fr**A**,fr**B




```
if ((frA) is a NaN or (frB) is a NaN)
       then c ← 0b0001
       else if (frA)< (frB)
                 then c ← 0b1000
                 else if (frA)> (frB)
                           then c ← 0b0100
                           else c ← 0b0010
FPCC ← c
CR[(4 * crfD)-(4 * crfD + 3)] ← c

if ((frA) is an SNaN or (frB) is an SNaN )
       then VXSNAN ← 1
if VE = 0
       then VXVC ← 1
       else if ((frA) is a QNaN or (frB) is a QNaN )
                 then VXVC ← 1
```

The floating-point operand in **fr**A is compared to the floating-point operand in **fr**B. The result of the compare is placed into CR field **crf**D and the FPCC.

If one of the operands is a NaN, either quiet or signaling, then CR field **crf**D and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, then VXSNAN is set, and if invalid operation is disabled (VE = 0) then VXVC is set. Otherwise, if one of the operands is a QNaN, then VXVC is set.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):

  Affected: LT, GT, EQ, UN

- Floating-Point Status and Control Register:

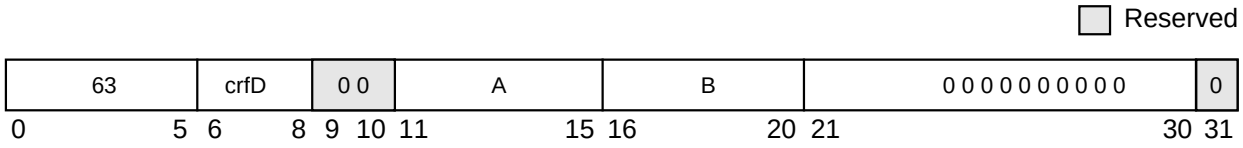
  Affected: FPCC, FX, VXSNAN, VXVC

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

fcmpu fcmpu

Floating Compare Unordered (x'FC00 0000')

fcmpu **crf**D**,fr**A**,fr**B




```
if ((frA) is a NaN or (frB) is a NaN)
      then c ← 0b0001
      else if (frA) < (frB)
            then c ← 0b1000
            else if (frA) > (frB)
                  then c ← 0b0100
                  else c ← 0b0010
FPCC ← c
CR[(4 * crfD)-(4 * crfD + 3)] ← c

if ((frA) is an SNaN or (frB) is an SNaN)
      then VXSNAN ← 1
```

The floating-point operand in register **fr**A is compared to the floating-point operand in register **fr**B. The result of the compare is placed into CR field **crf**D and the FPCC.

If one of the operands is a NaN, either quiet or signaling, then CR field **crf**D and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, then VXSNAN is set.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):

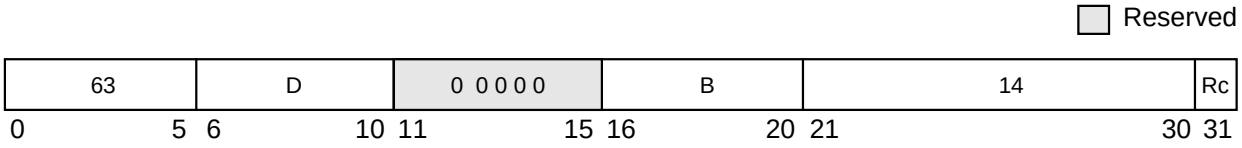
  Affected: LT, GT, EQ, UN

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

fctiw*x* fctiw*x*

Floating Convert to Integer Word (x'FC00 001C')

fctiw **fr**D**,fr**B (Rc = 0)
fctiw. **fr**D**,fr**B (Rc = 1)



The floating-point operand in register **fr**B is converted to a 32-bit signed integer, using the rounding mode specified by FPSCR[RN], and placed in bits 32–63 of **fr**D. Bits 0–31 of **fr**D are undefined.

If the operand in **fr**B are greater than $2^{31} - 1$, bits 32–63 of **fr**D are set to 0x7FFF_FFFF.

If the operand in **fr**B are less than -2^{31}, bits 32–63 of **fr**D are set to 0x8000_0000.

The conversion is described fully in Section D.4.2, "Floating-Point Convert to Integer Model," *in* the *PowerPC Microprocessor Family: The Programming Environments* manual.

Except for trap-enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

Do not use this instruction if the floating point register contains paired-single formatted data.

(programmers note: A **stiwz** instruction should be used to store the 32 bit resultant integer because bits 0–31 of **fr**D are undefined. A store double-precision instruction, e.g., **stfd**, will store the 64 bit result but 4 superfluous bytes are stored (bits **fr**D[0-31]). This may cause wasted bus traffic.)

Other registers altered:

- Condition Register (CR1 field):

  Affected: FX, FEX, VX, OX (if Rc = 1)

- Floating-Point Status and Control Register:

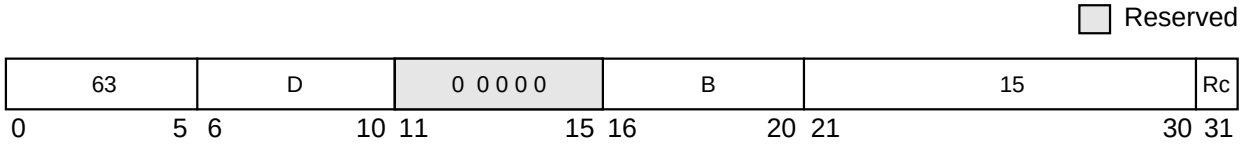
  Affected: FPRF (undefined), FR, FI, FX, XX, VXSNAN, VXCVI

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

fctiwz*x* fctiwz*x*

Floating Convert to Integer Word with Round toward Zero (x'FC00 001E')

fctiwz **fr**D,**fr**B (Rc = 0)
fctiwz. **fr**D,**fr**B (Rc = 1)



The floating-point operand in register **fr**B is converted to a 32-bit signed integer, using the rounding mode round toward zero, and placed in bits 32–63 of **fr**D. Bits 0–31 of **fr**D are undefined.

If the operand in **fr**B is greater than $2^{31} - 1$, bits 32–63 of **fr**D are set to 0x7FFF_FFFF.

If the operand in **fr**B is less than -2^{31}, bits 32–63 of **fr**D are set to 0x 8000_0000.

The conversion is described fully in Section D.4.2, "Floating-Point Convert to Integer Model" in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Except for trap-enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

Do not use this instruction if the floating point register contains paired-single formatted data.

(Programmers Note: A **stiwz** instruction should be used to store the 32 bit resultant integer because bits 0–31 of **fr**D are undefined. A store double-precision instruction, e.g., **stfd**, will store the 64 bit result but 4 superfluous bytes are stored (bits **fr**D[0-31]). This may cause wasted bus traffic.)

Other registers altered:

- Condition Register (CR1 field):

  Affected: FX, FEX, VX, OX (if Rc = 1)

- Floating-Point Status and Control Register:

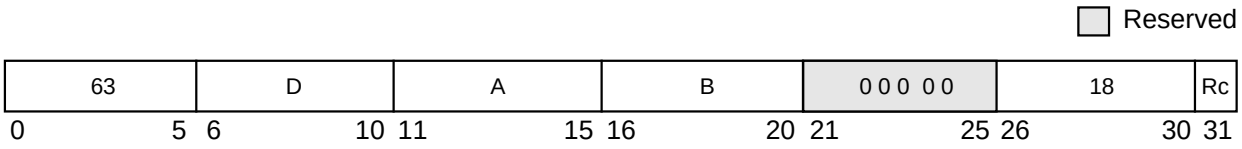
  Affected: FPRF (undefined), FR, FI, FX, XX, VXSNAN, VXCVI

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

fdivx fdivx

Floaiting Divide (Double-Precision),(x'FC00 0024')

fdiv **fr**D**,fr**A**,fr**B (Rc = 0)
fdiv. **fr**D**,fr**A**,fr**B (Rc = 1)



The floating-point operand in register **fr**A is divided by the floating-point operand in register **fr**B. The remainder is not supplied as a result.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

Floating-point division is based on exponent subtraction and division of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1 and zero divide exceptions when FPSCR[ZE] = 1.

Other registers altered:

- Condition Register (CR1 field):

  Affected: FX, FEX, VX, OX(if Rc = 1)
- Floating-Point Status and Control Register:

  Affected: FPRF, FR, FI, FX, OX, UX, ZX, XX, VXSNAN, VXIDI, VXZDZ

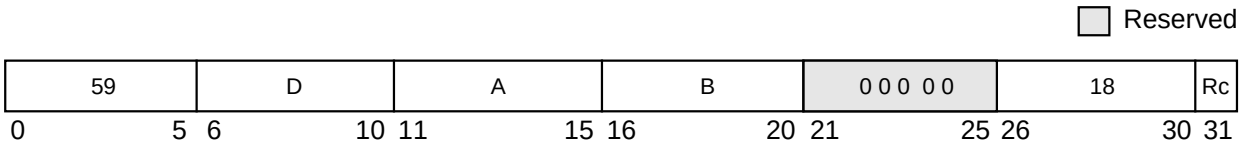
PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

fdivs*x* fdivs*x*

Floating Divide Single (x’EC00 0024’)

fdivs **fr**D**,fr**A**,fr**B (Rc = 0)

fdivs. **fr**D**,fr**A**,fr**B (Rc = 1)



The floating-point operand in register **fr**A is divided by the floating-point operand in register **fr**B. The remainder is not supplied as a result.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

Floating-point division is based on exponent subtraction and division of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1 and zero divide exceptions when FPSCR[ZE] = 1.

If the HID2[PSE] = 1 then the quotient is placed in both frD(ps0) and frD(ps1).

Other registers altered:

- Condition Register (CR1 field):

  Affected: FX, FEX, VX, OX(if Rc = 1)
- Floating-Point Status and Control Register:

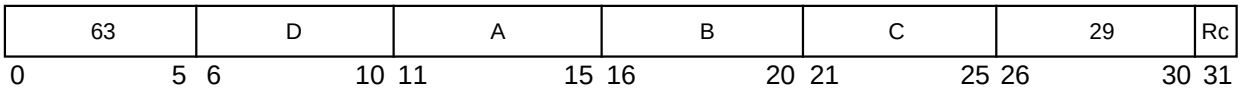
  Affected: FPRF, FR, FI, FX, OX, UX, ZX, XX, VXSNAN, VXIDI, VXZDZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

fmaddx fmaddx

Floating Multiply-Add (Double-Precision),(x'FC00 003A')

fmadd **fr**D**,fr**A**,fr**C**,fr**B (Rc = 0)
fmadd. **fr**D**,fr**A**,fr**C**,fr**B (Rc = 1)



The following operation is performed:

```
frD ← (frA * frC) + frB
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered:

- Condition Register (CR1 field):

  Affected: FX, FEX, VX, OX(if Rc = 1)
- Floating-Point Status and Control Register:

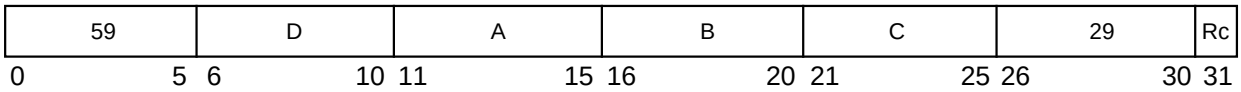
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

fmadds*x* fmadds*x*

Floaiting Multiply-Add Single (x'EC00 003A')

fmadds **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
fmadds. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The followings operation are performed:

```
if HID2[PSE] = 0
     then frD ← (frA * frC) + frB
     else frD(ps0) ← (frA(ps0) * frC(ps0)) + frB(ps0)
        frD(ps1) ← frD(ps0)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

If the HID2[PSE] = 1 then the result is placed in both **fr**D(ps0) and **fr**D(ps1).

Other registers altered:

- Condition Register (CR1 field):

  Affected: FX, FEX, VX, OX(if Rc = 1)

- Floating-Point Status and Control Register:

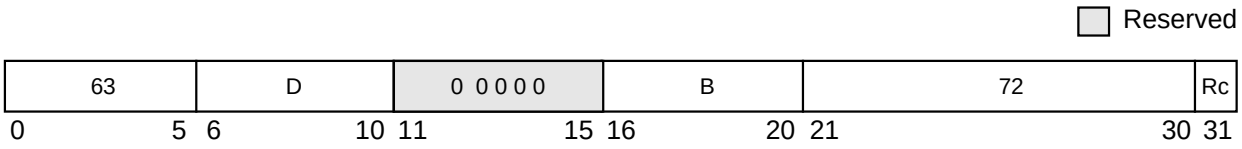
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

fmr*x* fmr*x*

Floating Move Register(Double-Precision),(x'FC00 0090')

fmr **fr**D**,fr**B (Rc = 0)
fmr. **fr**D**,fr**B (Rc = 1)



The content of register **fr**B is placed into **fr**D.

When HID2[PSE] = 1 and the content in **fr**B is a double-precision floating point operand, then the operand is copied to **fr**D.

When HID2[PSE] = 1 and the content of **fr**B contains a paired-single floating-point operand, the **fr**B[ps0] is copied to **fr**D[ps0] and the content of **fr**D[ps1] is unchanged.

Other registers altered:

- Condition Register (CR1 field):

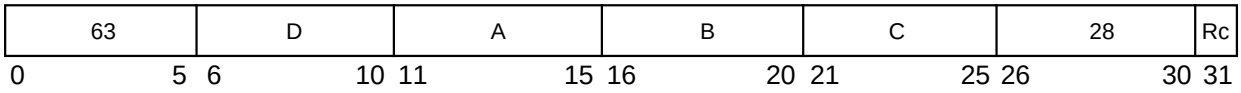
  Affected: FX, FEX, VX, OX(if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

fmsub*x* fmsub*x*

Floating Multiply-Subtract (Double-Precision),(x'FC00 0038')

fmsub **fr**D**,fr**A**,fr**C**,fr**B (Rc = 0)
fmsub. **fr**D**,fr**A**,fr**C**,fr**B (Rc = 1)



The following operation is performed:

```
frD ← [frA * frC] – frB
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered:

- Condition Register (CR1 field):

  Affected: FX, FEX, VX, OX(if Rc = 1)
- Floating-Point Status and Control Register:

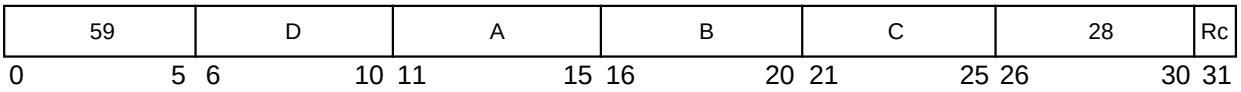
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

fmsubs*x* fmsubs*x*

Floating Multiply-Subtact Single (x'EC00 0038')

fmsubs **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
fmsubs. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0
then frD ← [frA * frC] – frB
else frD(ps0) ← [frA(ps0) * frC(ps0)] - frB(ps0)
       frD(ps1) ← frD(ps0)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

If the HID2[PSE] = 1 then the result is placed in both **fr**D(ps0) and **fr**D(ps1).

Other registers altered:

- Condition Register (CR1 field):

  Affected: FX, FEX, VX, OX(if Rc = 1)

- Floating-Point Status and Control Register:

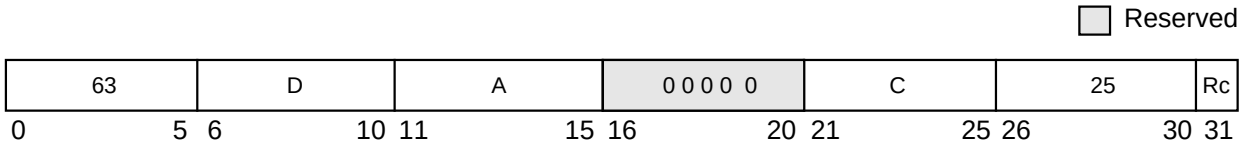
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

fmul*x* fmul*x*

Floating Multiply (Double-Precision),(x'FC00 0032')

fmul **fr**D**,fr**A**,fr**C (Rc = 0)
fmul. **fr**D**,fr**A**,fr**C (Rc = 1)



The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered:

- Condition Register (CR1 field):

  Affected: FX, FEX, VX, OX(if Rc = 1)
- Floating-Point Status and Control Register:

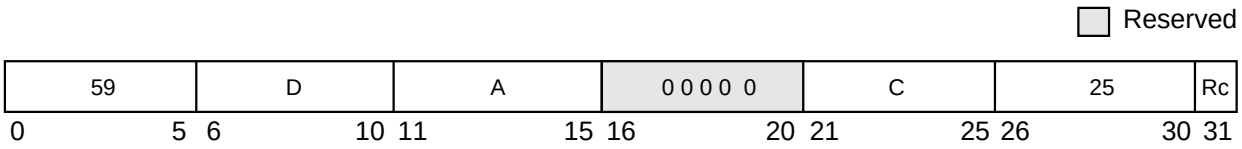
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXIMZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

fmuls*x* fmuls*x*

Floating Multiply Single (x'EC00 0032')

fmuls **fr**D**,fr**A**,fr**C (Rc = 0)
fmuls. **fr**D**,fr**A**,fr**C (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0
then frD ← frA * frC
else frD(ps0) ← frA(ps0) * frC(ps0)
       frD(ps1) ← frD(ps0)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

If the HID2[PSE] = 1 then the result is placed in both **fr**D(ps0) and **fr**D(ps1).

Other registers altered:

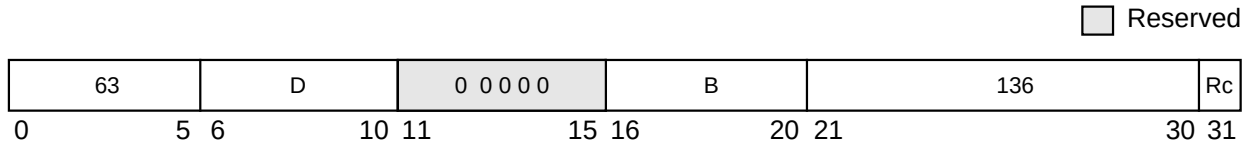
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXIMZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

fnabs*x* fnabs*x*

Floating Negative Absolute Value (x'FC00 0110')

fnabs **fr**D**,fr**B (Rc = 0)
fnabs. **fr**D**,fr**B (Rc = 1)



The contents of register **fr**B with bit 0 set are placed into **fr**D.

Note that the **fnabs** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN may be altered by **fnabs**. This instruction does not alter the FPSCR.

Other registers altered:

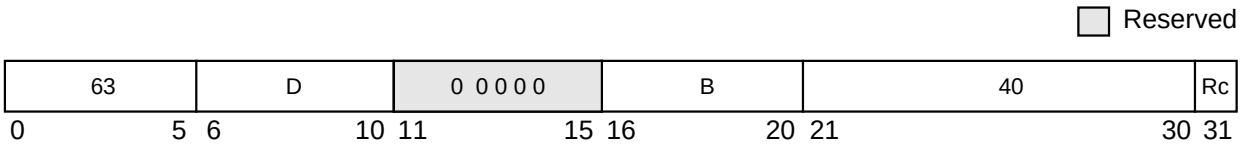
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

fneg*x* fneg*x*

Floating Negate (x'FC00 0050')

fneg **fr**D,**fr**B (Rc = 0)
fneg. **fr**D,**fr**B (Rc = 1)



The contents of register **fr**B with bit 0 inverted are placed into **fr**D.

Note that the **fneg** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN may be altered by **fneg**. This instruction does not alter the FPSCR.

Other registers altered:

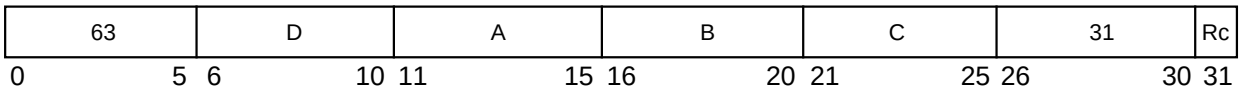
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

fnmadd*x* fnmadd*x*

Floating Negative Multiply-Add (Double-Precision),(x'FC00 003E')

fnmadd **fr**D**,fr**A**,fr**C**,fr**B (Rc = 0)
fnmadd. **fr**D**,fr**A**,fr**C**,fr**B (Rc = 1)



The following operation is performed:

```
frD ← – ([frA * frC] + frB)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **fr**D.

This instruction produces the same result as would be obtained by using the Floating Multiply-Add (**fmadd***x*) instruction and then negating the result, with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered:

- Condition Register (CR1 field):

  Affected: FX, FEX, VX, OX (if Rc = 1)

- Floating-Point Status and Control Register:

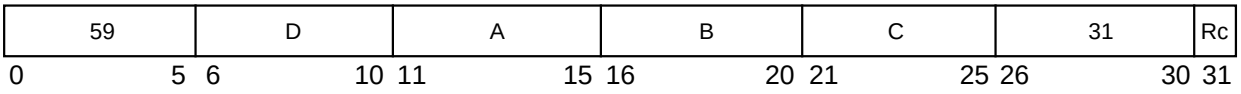
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

fnmadds*x* fnmadds*x*

Floating Negative Multiply-Add Single (x'EC00 003E')

fnmadds **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
fnmadds. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0
then  frD ← -([frA * frC] + frB)
else  frD(ps0) ← -([frA(ps0) * frC(ps0)] + frB(ps0))
      frD(ps1) ← frD(ps0)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **fr**D.

This instruction produces the same result as would be obtained by using the Floating Multiply-Add Single (**fmadds***x*) instruction and then negating the result, with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

If the HID2[PSE] = 1 then the result is placed in both **fr**D(ps0) and **fr**D(ps1).

Other registers altered:

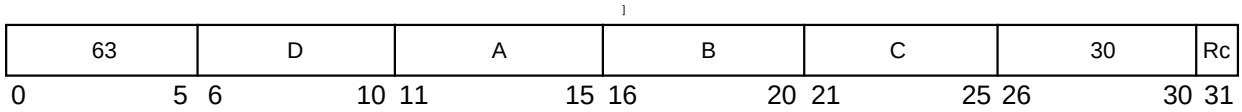
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

fnmsub*x* fnmsub*x*

Floating Negative Multiply-Subtract (Double-Precision),(x'FC00 003C')

fnmsub **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
fnmsub. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operation is performed:

```
frD ← – ([frA * frC] – frB)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result.

If the most-significant bit of the resultant significand is not one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **fr**D.

This instruction produces the same result obtained by negating the result of a Floating Multiply-Subtract (**fmsub***x*) instruction with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered:

- Condition Register (CR1 field)

  Affected: FX, FEX, VX, OX(if Rc = 1)

- Floating-Point Status and Control Register:

  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

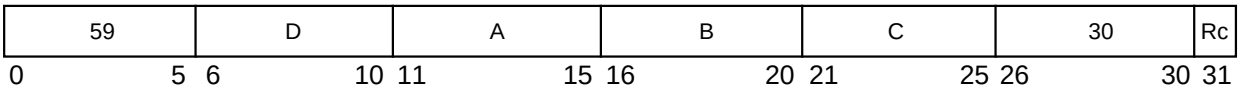
PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

fnmsubs*x* fnmsubs*x*

Floating Negative Multiply-Subtract Single (x'EC00 003C')

fnmsubs **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)

fnmsubs. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0
then   frD ← -([frA * frC] - frB)
else   frD(ps0) ← -([frA(ps0) * frC(ps0)] - frB(ps0))
       frD(ps1) ← frD(ps0)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result.

If the most-significant bit of the resultant significand is not one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **fr**D.

This instruction produces the same result obtained by negating the result of a Floating Multiply-Subtract Single (**fmsubs***x*) instruction with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

If the HID2[PSE] = 1 then the result is placed in both **fr**D(ps0) and **fr**D(ps1).

Other registers altered:

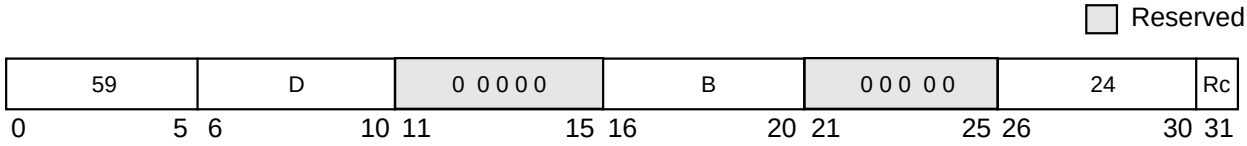
- Condition Register (CR1 field)
  Affected: FX, FEX, VX, OX(if Rc = 1)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

fres*x* fres*x*

Floating Reciprocal Estimate Single (x'EC00 0030')

fres	**fr**D,**fr**B	(Rc = 0)
fres.	**fr**D,**fr**B	(Rc = 1)




```
if HID2[PSE] = 0
then   frD ← estimate[ 1/frB]
else   frD(ps0) ← estimate[ 1/frB(ps0)]
       frD(ps1) ← frD(ps0)
```

A single-precision estimate of the reciprocal of the floating-point operand in register **fr**B is placed into register **fr**D. The estimate placed into register **fr**D is correct to a precision of one part in 4096 of the reciprocal of **fr**B. That is,

$$\mathrm{ABS}\left(\frac{\text{estimate}-\left(\frac{1}{x}\right)}{\left(\frac{1}{x}\right)}\right) \le \frac{1}{(4096)}$$

where x is the initial value in **fr**B. Note that the value placed into register **fr**D may vary between implementations, and between different executions on the same implementation.

Operation with various special values of the operand is summarized below:

Operand	Result	Exception
–	–0	None
–0	– *	ZX
+0	+ *	ZX
+	+0	None
SNaN	QNaN**	VXSNAN
QNaN	QNaN	None

Notes: * No result if FPSCR[ZE] = 1

** No result if FPSCR[VE] = 1

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1 and zero divide exceptions when FPSCR[ZE] = 1.

NOTE: The PowerPC architecture makes no provision for a double-precision version of the **fres***x* instruction. This is because graphics applications are expected to need only the single-precision version, and no other important performance-critical applications are expected to require a double-precision version of the **fres***x* instruction.

If the HID2[PSE] = 1 then the result is placed in both **fr**D(ps0) and **fr**D(ps1).

This instruction is optional in the PowerPC architecture.Other registers altered:

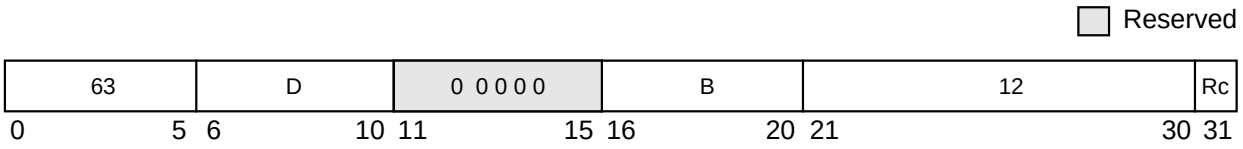
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR (undefined), FI (undefined), FX, OX, UX, ZX, VXSNAN

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA			YES	A

frsp*x* frsp*x*

Floating Round to Single (x'FC00 0018')

frsp **fr**D**,fr**B (Rc = 0)
frsp. **fr**D**,fr**B (Rc = 1)



If HID2[PSE] = 0 then the floating-point operand in register **fr**B is rounded to single-precision using the rounding mode specified by FPSCR[RN] and placed into **fr**D.

If HID2[PSE] = 1 then the source operand in register **fr**B is rounded to single-precision using the rounding mode specified by FPSCR[RN] and placed into **fr**D(ps0). The value in **fr**D(ps1) is undefined.

The rounding is described fully in Section D.4.1, "Floating-Point Round to Single-Precision Model," in *The Programming Environments Manual.*

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered:

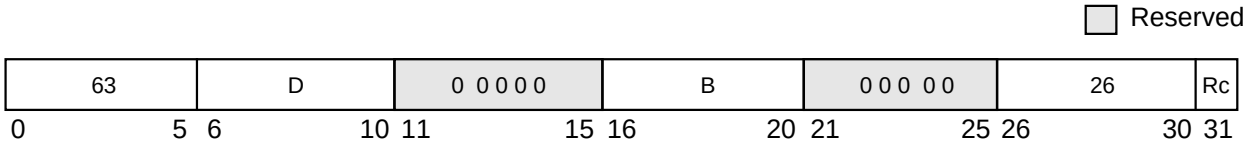
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX(if Rc = 1)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

frsqrte*x* frsqrte*x*

Floating Reciprocal Square Root Estimate (x'FC00 0034')

frsqrte	**fr**D,**fr**B	(Rc = 0)
frsqrte.	**fr**D,**fr**B	(Rc = 1)



A double-precision estimate of the reciprocal of the square root of the floating-point operand in register **fr**B is placed into register **fr**D. The estimate placed into register **fr**D is correct to a precision of one part in 4096 of the reciprocal of the square root of **fr**B. That is,

$$\mathrm{ABS}\left(\frac{\text{estimate}-\left(\frac{1}{\sqrt{x}}\right)}{\left(\frac{1}{\sqrt{x}}\right)}\right) \le \frac{1}{4096}$$

where x is the initial value in **fr**B. Note that the value placed into register **fr**D may vary between implementations, and between different executions on the same implementation.

Operation with various special values of the operand is summarized below:

Operand	Result	Exception
–	QNaN**	VXSQRT
<0	QNaN**	VXSQRT
–0	– *	ZX
+0	+ *	ZX
+	+0	None
SNaN	QNaN**	VXSNAN
QNaN	QNaN	None

Notes: * No result if FPSCR[ZE] = 1

** No result if FPSCR[VE] = 1

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1 and zero divide exceptions when FPSCR[ZE] = 1.

NOTE: No single-precision version of the **frsqrte** instruction is provided; however, both **fr**B and **fr**D are representable in single-precision format.

This instruction is optional in the PowerPC architecture.

Other registers altered:

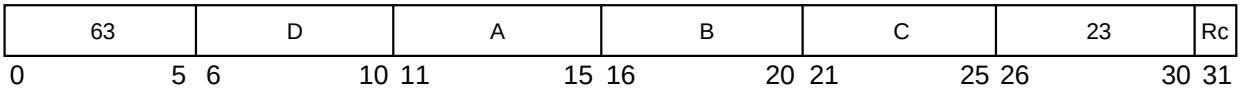
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX(if Rc = 1)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR (undefined), FI (undefined), FX, ZX, VXSNAN, VXSQRT

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA			Yes	A

fsel*x* fsel*x*

Floating Select (x'FC00 002E')

fsel **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
fsel. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)




```
if (frA) ≥ 0.0
then frD ← (frC)
else frD ← (frB)
```

The floating-point operand in register **fr**A is compared to the value zero. If the operand is greater than or equal to zero, register **fr**D is set to the contents of register **fr**C. If the operand is less than zero or is a NaN, register **fr**D is set to the contents of register **fr**B. The comparison ignores the sign of zero (that is, regards +0 as equal to –0).

Care must be taken in using **fsel** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities.

For examples of uses of this instruction, see Section D.3, "Floating-Point Conversions," and Section D.5, "Floating-Point Selection," in *The Programming Environments Manual.*

This instruction is optional in the PowerPC architecture.

When HID2[PSE] = 1 and the selected source is a double-precision floating-point operand, then the selected operand from **fr**B or **fr**C is copied to **fr**D (as described above).

When HID2[PSE] = 1 and the selected source contains paired-single floating-point operands, only **fr**A(ps0) is compared to zero and the selected operand from **fr**B(ps0) or **fr**C(ps0) is copied to **fr**D[ps0]. The content of **fr**D[ps1] is undefined.

Other registers altered:

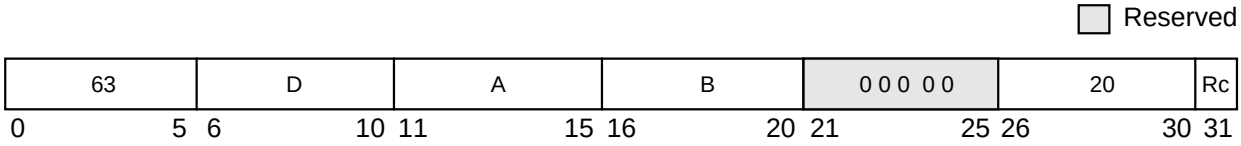
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX(if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA			Yes	A

fsub*x* fsub*x*

Floating Subtract (Double-Precision),(x'FC00 0028')

fsub **fr**D**,fr**A**,fr**B (Rc = 0)
fsub. **fr**D**,fr**A**,fr**B (Rc = 1)



The floating-point operand in register **fr**B is subtracted from the floating-point operand in register **fr**A. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

The execution of the **fsub** instruction is identical to that of **fadd**, except that the contents of **fr**B participate in the operation with its sign bit (bit 0) inverted.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered:

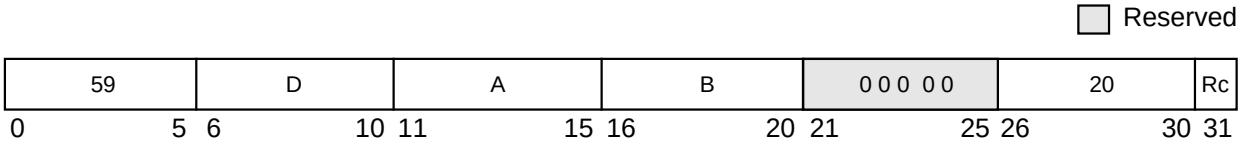
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX(if Rc = 1)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

fsubsx fsubs*x*

Floating Subtract Single (x'EC00 0028')

fsubs **fr**D**,fr**A**,fr**B (Rc = 0)
fsubs. **fr**D**,fr**A**,fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0
then  frD ← frA – frB
else  frD(ps0) ← frA(ps0) - frB(ps0)
      frD(ps1) ← frD(ps0)
```

The floating-point operand in register **fr**B is subtracted from the floating-point operand in register **fr**A. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

The execution of the **fsubs** instruction is identical to that of **fadds**, except that the contents of **fr**B participate in the operation with its sign bit (bit 0) inverted.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

If the HID2[PSE] = 1 then the result is placed in both **fr**D(ps0) and **fr**D(ps1).

Other registers altered:

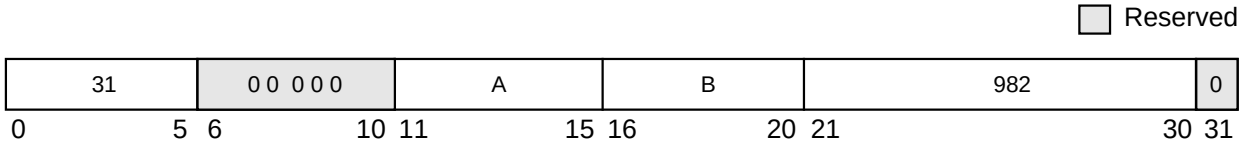
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control Register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				A

icbi icbi

Instruction Cache Block Invalidate (x'7C00 07AC')

icbi **r**A,**r**B



EA is the sum (**r**A|0) + (**r**B).

If the block containing the byte addressed by EA is in coherency-required mode, and a block containing the byte addressed by EA is in the instruction cache of any processor, the block is made invalid in all such instruction caches, so that subsequent references cause the block to be refetched.

If the block containing the byte addressed by EA is in coherency-not-required mode, and a block containing the byte addressed by EA is in the instruction cache of this processor, the block is made invalid in that instruction cache, so that subsequent references cause the block to be refetched.

The function of this instruction is independent of the write-through, write-back, and caching-inhibited/allowed modes of the block containing the byte addressed by EA.

This instruction is treated as a load from the addressed byte with respect to address translation and memory protection. It may also be treated as a load for referenced and changed bit recording except that referenced and changed bit recording may not occur. Implementations with a combined data and instruction cache treat the **icbi** instruction as a no-op, except that they may invalidate the target block in the instruction caches of other processors if the block is in coherency-required mode.The **icbi** instruction invalidates the block at EA (**r**A|0 + **r**B). If the processor is a multiprocessor implementation (for example, the 601, 604, or 620) and the block is marked coherency-required, the processor will send an address-only broadcast to other processors causing those processors to invalidate the block from their instruction caches.

For faster processing, many implementations will not compare the entire EA (**r**A|0 + **r**B) with the tag in the instruction cache. Instead, they will use the bits in the EA to locate the set that the block is in, and invalidate all blocks in that set.

Other registers altered:

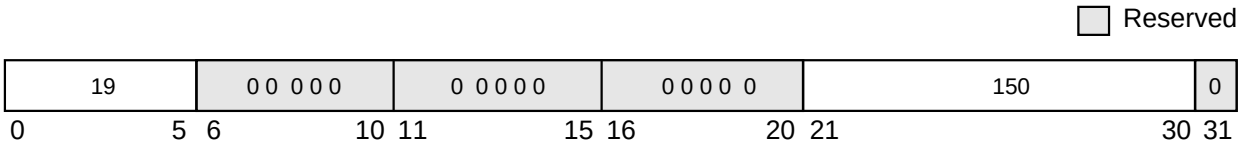
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
VEA				X

isync isync

Instruction Synchronize (x'4C00 012C')

isync



The **isync** instruction provides an ordering function for the effects of all instructions executed by a processor. Executing an **isync** instruction ensures that all instructions preceding the **isync** instruction have completed before the **isync** instruction completes, except that memory accesses caused by those instructions need not have been performed with respect to other processors and mechanisms. It also ensures that no subsequent instructions are initiated by the processor until after the **isync** instruction completes. Finally, it causes the processor to discard any prefetched instructions, with the effect that subsequent instructions will be fetched and executed in the context established by the instructions preceding the isync instruction. The **isync** instruction has no effect on the other processors or on their caches.

This instruction is context synchronizing.

Context synchronization is necessary after certain code sequences that perform complex operations within the processor. These code sequences are usually operating system tasks that involve memory management. For example, if an instruction A changes the memory translation rules in the memory management unit (MMU), the **isync** instruction should be executed so that the instructions following instruction A will be discarded from the pipeline and refetched according to the new translation rules.

NOTE: All exceptions and the **rfi** and **sc** instructions are also context synchronizing.

Other registers altered:

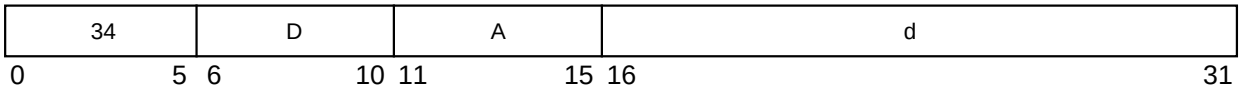
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
VEA				XL

lbz lbz

Load Byte and Zero (x’8800 0000’)

lbz **r**D**,**d**(r**A**)**




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
rD ← (24)0 || MEM(EA, 1)
```

EA is the sum (**r**A|0) + d. The byte in memory addressed by EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared.

Other registers altered:

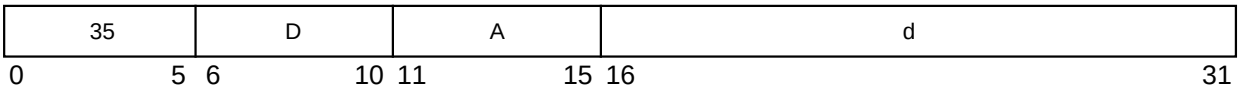
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

lbzu lbzu

Load Byte and Zero with Update (x'8C00 0000')

lbzu **r**D,d(**r**A)




```
EA ← (rA) + EXTS(d)
rD← (24)0 || MEM(EA, 1)
rA← EA
```

EA is the sum (**r**A) + d. The byte in memory addressed by EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared.

EA is placed into **r**A.

If **r**A = 0, or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

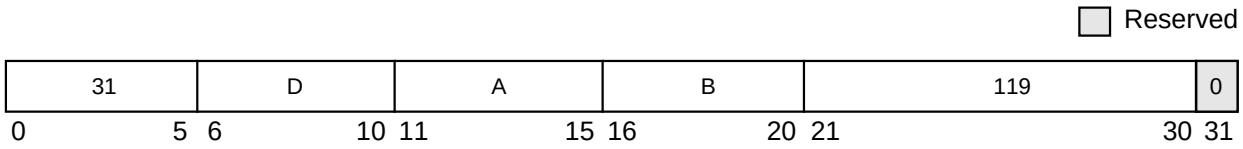
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

lbzux lbzux

Load Byte and Zero with Update Indexed (x'7C00 00EE')

lbzux **r**D,**r**A,**r**B




```
EA ← (rA) + (rB)
rD ← (24)0 || MEM(EA, 1)
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The byte in memory addressed by EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

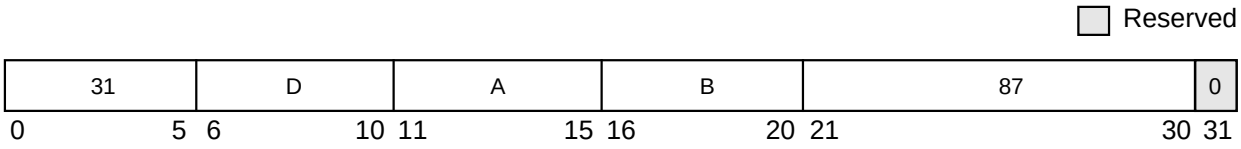
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lbzx lbzx

Load Byte and Zero Indexed (x'7C00 00AE')

lbzx **r**D,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
rD ← (24)0 || MEM(EA, 1)
```

EA is the sum (**r**A|0) + (**r**B). The byte in memory addressed by EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared.

Other registers altered:

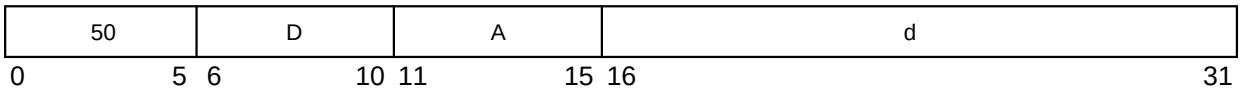
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lfd lfd

Load Floating-Point Double (x'C800 0000')

lfd **fr**D,d**(r**A**)**




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
frD ← MEM(EA, 8)
```

EA is the sum (**r**A|0) + d.

The double word in memory addressed by EA is placed into **fr**D.

Other registers altered:

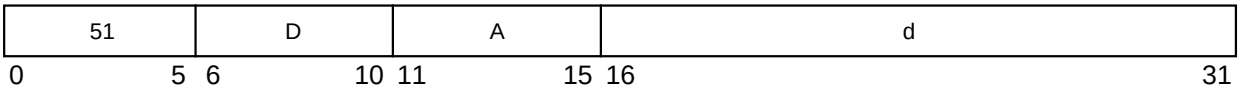
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

lfdu lfdu

Load Floating-Point Double with Update (x'CC00 0000')

lfdu **fr**D,d**(r**A**)**




```
EA ← (rA) + EXTS(d)
frD ← MEM(EA, 8)
rA ← EA
```

EA is the sum (**r**A) + d.

The double word in memory addressed by EA is placed into **fr**D.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

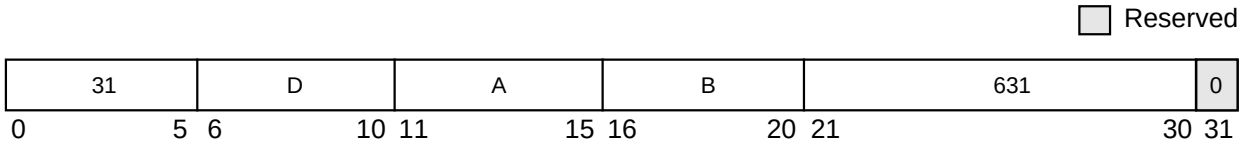
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

lfdux lfdux

Load Floating-Point Double with Update Indexed (x'7C00 04EE')

lfdux **fr**D,**r**A,**r**B




```
EA ← (rA) + (rB)
frD ←MEM(EA, 8)
rA ← EA
```

EA is the sum (**r**A) + (**r**B).

The double word in memory addressed by EA is placed into **fr**D.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

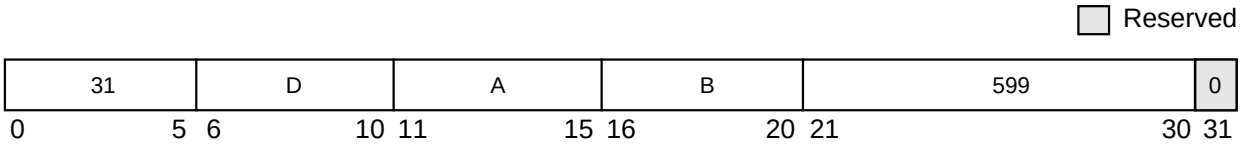
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lfdx lfdx

Load Floating-Point Double Indexed (x'7C00 04AE')

lfdx **fr**D,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
frD ← MEM(EA, 8)
```

EA is the sum (**r**A|0) + (**r**B).

The double word in memory addressed by EA is placed into **fr**D.

Other registers altered:

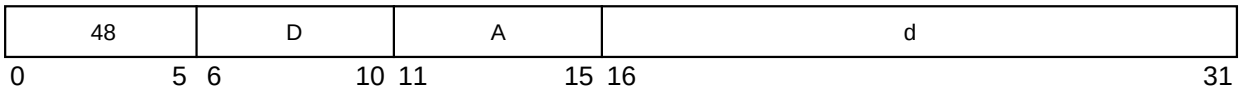
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lfs lfs

Load Floating-Point Single (x'C000 0000')

lfs **fr**D,d**(r**A**)**




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
if HID2[PSE] = 0
 then frD ← DOUBLE(MEM(EA, 4))
 else frD(ps0) ← Single(MEM(EA, 4))
      frD(ps1) ← Single(MEM(EA, 4))
```

The word in memory addressed by EA is interpreted as a floating-point single-precision operand.

If HID2[PSE] = 0 then this word is converted to floating-point double-precision and placed into **fr**D.

If HID2[PSE] = 1 then this word is interpreted as a floating-point single-precision operand and placed into **fr**D(ps0) and replicated in **fr**D(ps1).

Other registers altered:

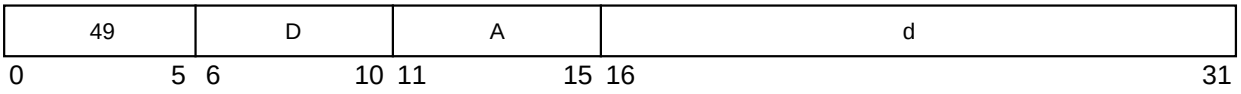
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

lfsu lfsu

Load Floating-Point Single with Update (x'C400 0000')

lfsu **fr**D,d**(r**A**)**




```
EA ← (rA) + EXTS(d)
rA ← EA
if HID2[PSE] = 0
then frD ← DOUBLE(MEM(EA, 4))
else frD(ps0) ← Single(MEM(EA, 4))
     frD(ps1) ← Single(MEM(EA, 4))
```

EA is the sum (**r**A) + d.

The word in memory addressed by EA is interpreted as a floating-point single-precision operand.

If HID2[PSE] = 0 then this word is converted to floating-point double-precision and placed into **fr**D.

If HID2[PSE] = 1 then this word is interpreted as a floating-point single-precision operand and placed into **fr**D(ps0) and replicated in **fr**D(ps1).

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

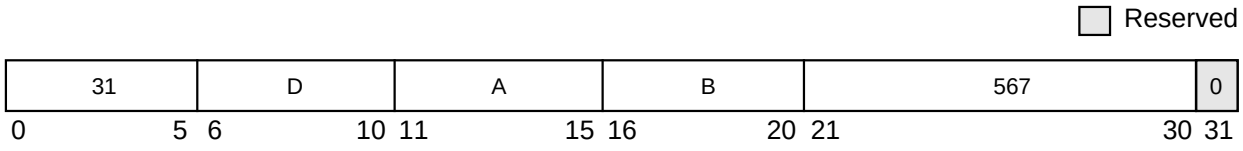
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

lfsux lfsux

Load Floating-Point Single with Update Indexed (x'7C00 046E')

lfsux **fr**D,**r**A,**r**B




```
EA ← (rA) + (rB)
if HID2[PSE] = 0
then frD ← DOUBLE(MEM(EA, 4))
else frD(ps0) ← Single(MEM(EA, 4))
     frD(ps1) ← Single(MEM(EA, 4))
rA ← EA
```

EA is the sum (**r**A) + d.

The word in memory addressed by EA is interpreted as a floating-point single-precision operand.

If HID2[PSE] = 0 then this word is converted to floating-point double-precision (see Section D.6, "Floating-Point Load Instructions," in *The Programming Environments Manual*) and placed into **fr**D.

If HID2[PSE] = 1 then this word is interpreted as a floating-point single-precision operand and placed into **fr**D(ps0) and replicated in **fr**D(ps1).

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

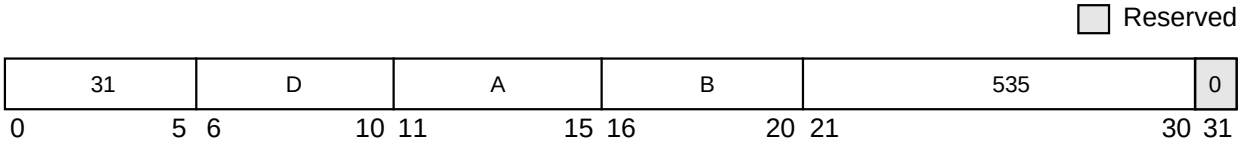
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lfsx lfsx

Load Floating-Point Single Indexed (x'7C00 042E')

lfsx **fr**D,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
if HID2[PSE] = 0
then frD ← DOUBLE(MEM(EA, 4))
else frD(ps0) ← Single(MEM(EA, 4))
     frD(ps1) ← Single(MEM(EA, 4))
```

EA is the sum (**r**A|0) + (rB).

The word in memory addressed by EA is interpreted as a floating-point single-precision operand.

If HID2[PSE] = 0 then this word is converted to floating-point double-precision and placed into **fr**D.

If HID2[PSE] = 1 then this word is interpreted as a floating-point single-precision operand and placed into **fr**D(ps0) and replicated in **fr**D(ps1).

Other registers altered:

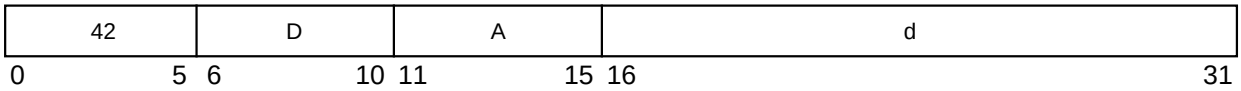
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lha lha

Load Half Word Algebraic (x'A800 0000')

lha **r**D,d**(r**A**)**




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
rD ← EXTS(MEM(EA, 2))
```

EA is the sum (**r**A|0) + d. The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most-significant bit of the loaded half word.

Other registers altered:

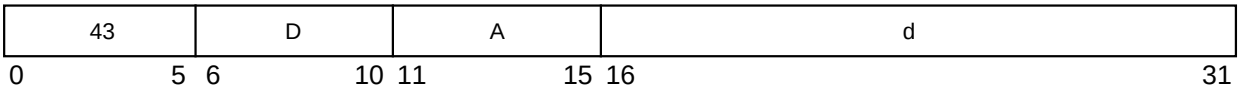
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

lhau lhau

Load Half Word Algebraic with Update (x'AC00 0000')

lhau **r**D,d**(r**A**)**




```
EA ← (rA) + EXTS(d)
rD ← EXTS(MEM(EA, 2))
rA ← EA
```

EA is the sum (**r**A) + d. The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most-significant bit of the loaded half word.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

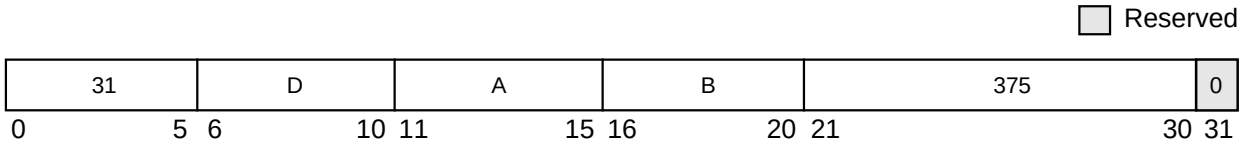
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

lhaux lhaux

Load Half Word Algebraic with Update Indexed (x'7C00 02EE')

lhaux **r**D,**r**A,**r**B




```
EA ← (rA) + (rB)
rD ← EXTS(MEM(EA, 2))
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most-significant bit of the loaded half word.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

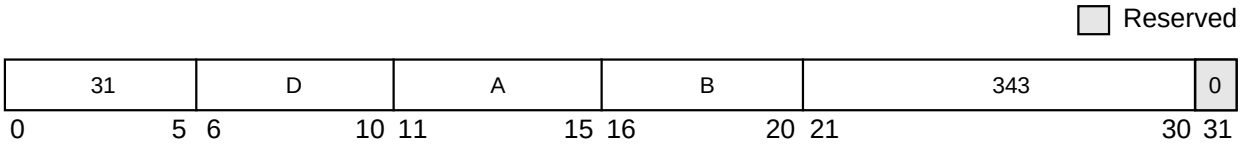
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lhax lhax

Load Half Word Algebraic Indexed (x'7C00 02AE')

lhax **r**D,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
rD ← EXTS(MEM(EA, 2)
```

EA is the sum (**r**A|0) + (**r**B). The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most-significant bit of the loaded half word.

Other registers altered:

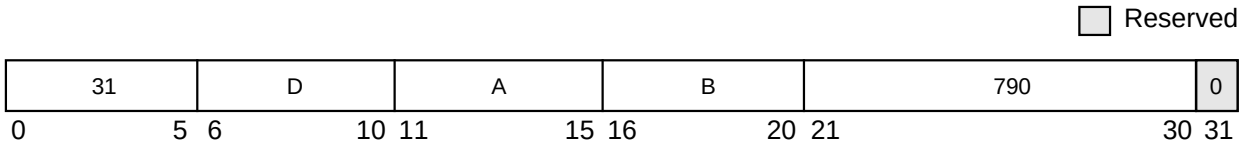
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lhbrx lhbrx

Load Half Word Byte-Reverse Indexed (x'7C00 062C')

lhbrx **r**D,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
rD ← (16)0 || MEM(EA + 1, 1) || MEM(EA, 1)
```

EA is the sum (**r**A|0) + (**r**B). Bits 0–7 of the half word in memory addressed by EA are loaded into the low-order eight bits of **r**D. Bits 8–15 of the half word in memory addressed by EA are loaded into the subsequent low-order eight bits of **r**D. The remaining bits in **r**D are cleared.

The PowerPC architecture cautions programmers that some implementations of the architecture may run the **lhbrx** instructions with greater latency than other types of load instructions.

Other registers altered:

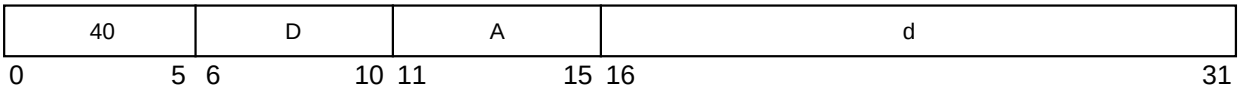
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lhz lhz

Load Half Word and Zero (x'A000 0000')

lhz **r**D,d**(r**A**)**




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
rD ← (16)0 || MEM(EA, 2)
```

EA is the sum (**r**A|0) + d. The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.

Other registers altered:

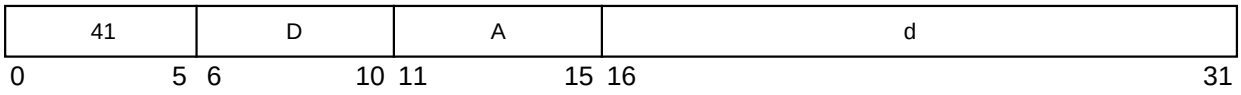
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

lhzu lhzu

Load Half Word and Zero with Update (x'A400 0000')

lhzu **r**D,d**(r**A**)**




```
EA ← rA + EXTS(d)
rD ← (16)0 || MEM(EA, 2)
rA ← EA
```

EA is the sum (**r**A) + d. The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

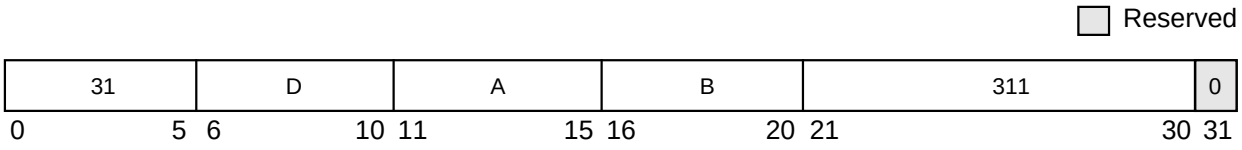
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

lhzux lhzux

Load Half Word and Zero with Update Indexed (x'7C00 026E')

lhzux **r**D,**r**A,**r**B




```
EA ← (rA) + (rB)
rD← (16)0 || MEM(EA, 2)
rA← EA
```

EA is the sum (**r**A) + (**r**B). The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

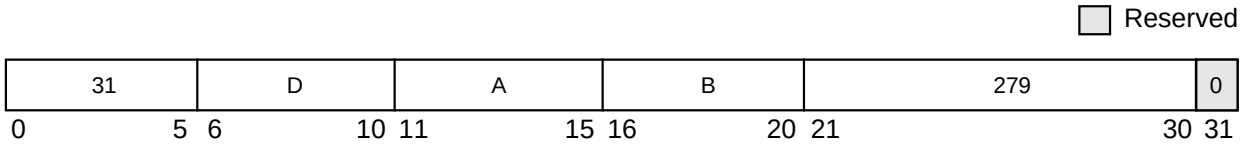
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lhzx lhzx

Load Half Word and Zero Indexed (x'7C00 022E')

lhzx **r**D,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
rD ← (16)0 || MEM(EA, 2)
```

EA is the sum (**r**A|0) + (**r**B). The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.

Other registers altered:

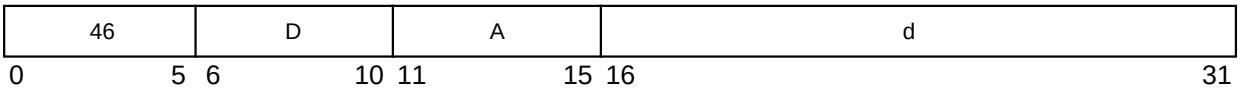
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lmw lmw

Load Multiple Word (x'B800 0000')

lmw **r**D,d**(r**A**)**




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
r ← rD
do while r ≤ 31
GPR(r) ← MEM(EA, 4)
r ← r + 1
EA ← EA + 4
```

EA is the sum (**r**A|0) + d.

n = (32 – **r**D).

n consecutive words starting at EA are loaded into GPRs **r**D through **r31**.

EA must be a multiple of four. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI exceptions, see Section 6.4.3, "DSI Exception (0x00300)," in *The Programming Environments Manual*.

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

Note that, in some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Other registers altered:

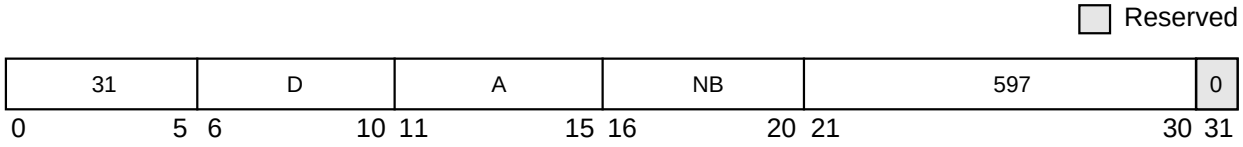
None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

lswi lswi

Load String Word Immediate (x'7C00 04AA')

lswi **r**D**,r**A**,**NB




```
if rA = 0
then EA ← 0
else EA ← (rA)
if NB = 0
then n ← 32
else n ← NB
r ← rD – 1
i ← 0
do while n > 0
     if i = 0
        then r ← r + 1 (mod 32)
          GPR(r) ← 0
     GPR(r)[i,i + 7] ← MEM(EA, 1)
     i ← i + 8
     if i = 32 then i ← 0
     EA ← EA + 1
     n ← n – 1
```

EA is (**r**A | 0).

Let *n* = NB if NB 0,*n* = 32 if NB = 0; *n* is the number of bytes to load.

Let *nr* = CEIL(*n* , 4);*nr* is the number of registers to be loaded with data.

n consecutive bytes starting at EA are loaded into GPRs **r**D through **r**D + *nr* – 1.

Bytes are loaded left to right in each register. The sequence of registers wraps around to **r0** if required. If the 4 bytes of register **r**D + *nr* – 1 are only partially filled, the unfilled low-order byte(s) of that register are cleared.

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

Under certain conditions (for example, segment boundary crossing) the data alignment exception handler may be invoked. For additional information about data alignment exceptions, see Section 6.4.3, "DSI Exception (0x00300)," in *The Programming Environments Manual*. Note that, in some implementations, this instruction is likely to have greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Other registers altered:

- None

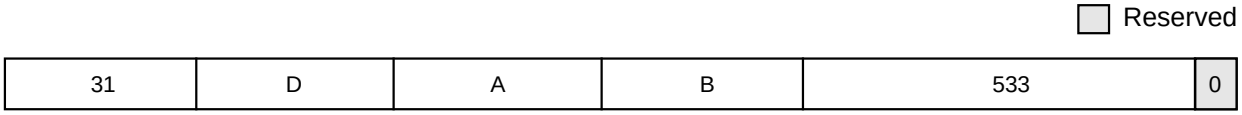
PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lswx

lswx

Load String Word Indexed (x'7C00 042A')

lswx **r**D,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
n ← XER[25–31]
r ← rD – 1
i ← 0
rD ← undefined
 do while n > 0
      if i = 0
         then r ← r + 1 (mod 32)
           GPR(r) ← (32)0
      GPR(r)[i,i + 7] ← MEM(EA, 1)
      i ← i + 8
      if i = 32 then i ← 0
      EA ← EA + 1
      n ← n – 1
```

EA is the sum (**r**A|0) + (**r**B). Let *n* = XER[25–31]; *n* is the number of bytes to load. Let *nr* = CEIL(*n* 4); *nr* is the number of registers to receive data. If *n* > 0, *n* consecutive bytes starting at EA are loaded into GPRs **r**D through **r**D + *nr* – 1. Bytes are loaded left to right in each register. The sequence of registers wraps around through **r0** if required. If the four bytes of **r**D + *nr* – 1 are only partially filled, the unfilled low-order byte(s) of that register are cleared. If *n* = 0, the contents of **r**D are undefined.

If **r**A or **r**B is in the range of registers specified to be loaded, including the case in which **r**A = 0, either the system illegal instruction error handler is invoked or the results are boundedly undefined. If **r**D = **r**A or **r**D = **r**B, the instruction form is invalid; If **r**D and **r**A both specify GPR0, the form is invalid.

Under certain conditions (for example, segment boundary crossing) the data alignment exception handler may be invoked. For additional information about data alignment exceptions, see Section 6.4.3, "DSI Exception (0x00300)," in *The Programming Environments Manual.*

NOTE: In some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

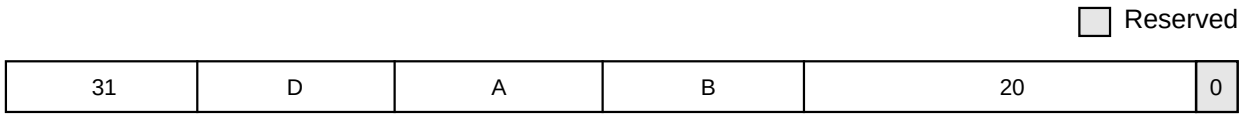
Other registers altered: None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lwarx lwarx

Load Word and Reserve Indexed (x'7C00 0028')

lwarx **r**D,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
RESERVE ← 1
RESERVE_ADDR ← physical_addr(EA)
rD ← MEM(EA,4)
```

EA is the sum (**r**A|0) + (**r**B).

The word in memory addressed by EA is loaded into **r**D.

This instruction creates a reservation for use by a store word conditional indexed (**stwcx.**)instruction. The physical address computed from EA is associated with the reservation, and replaces any address previously associated with the reservation.

EA must be a multiple of four. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI exceptions, see Section 6.4.3, "DSI Exception (0x00300)," in *The Programming Environments Manual.*

When the RESERVE bit is set, the processor enables hardware snooping for the block of memory addressed by the RESERVE address. If the processor detects that another processor writes to the block of memory it has reserved, it clears the RESERVE bit. The **stwcx.** instruction will only do a store if the RESERVE bit is set. The **stwcx.** instruction sets the CR0[EQ] bit if the store was successful and clears it if it failed. The **lwarx** and **stwcx.** combination can be used for atomic read-modify-write sequences. Note that the atomic sequence is not guaranteed, but its failure can be detected if CR0[EQ] = 0 after the **stwcx.** instruction.

Other registers altered:

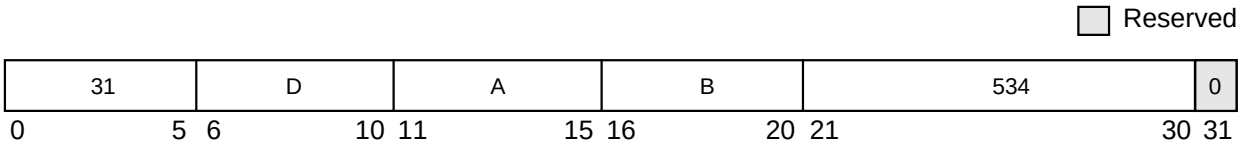
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lwbrx lwbrx

Load Word Byte-Reverse Indexed (x'7C00 042C')

lwbrx **r**D,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
rD ← MEM(EA + 3, 1) || MEM(EA + 2, 1) || MEM(EA + 1, 1) || MEM(EA, 1)
```

EA is the sum (**r**A|0) + **r**B. Bits 0–7 of the word in memory addressed by EA are loaded into the low-order 8 bits of **r**D. Bits 8–15 of the word in memory addressed by EA are loaded into the subsequent low-order 8 bits of **r**D. Bits 16–23 of the word in memory addressed by EA are loaded into the subsequent low-order eight bits of **r**D. Bits 24–31 of the word in memory addressed by EA are loaded into the subsequent low-order 8 bits of **r**D.

The PowerPC architecture cautions programmers that some implementations of the architecture may run the **lwbrx** instructions with greater latency than other types of load instructions.

Other registers altered:

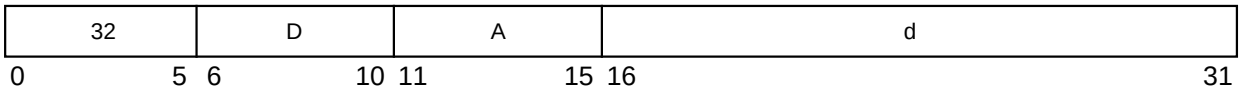
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lwz lwz

Load Word and Zero (x'8000 0000')

lwz **r**D,d**(r**A**)**




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
rD ← MEM(EA, 4)
```

EA is the sum (**r**A|0) + d. The word in memory addressed by EA is loaded into **r**D.

Other registers altered:

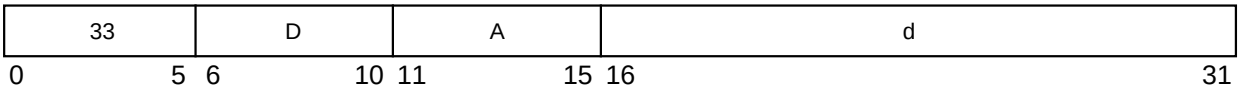
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

lwzu lwzu

Load Word and Zero with Update (x'8400 0000')

lwzu **r**D,d**(r**A**)**




```
EA ← rA + EXTS(d)
rD ← MEM(EA, 4)
rA ← EA
```

EA is the sum (**r**A) + d. The word in memory addressed by EA is loaded into **r**D.

EA is placed into **r**A.

If **r**A = 0, or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

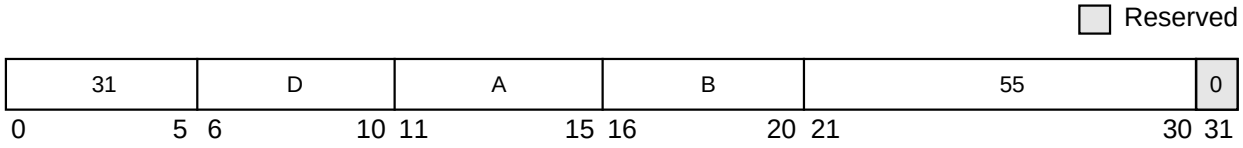
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

lwzux lwzux

Load Word and Zero with Update Indexed (x'7C00 006E')

lwzux **r**D,**r**A,**r**B




```
EA ← (rA) + (rB)
rD ← MEM(EA, 4)
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The word in memory addressed by EA is loaded into **r**D

EA is placed into **r**A.

If **r**A = 0, or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

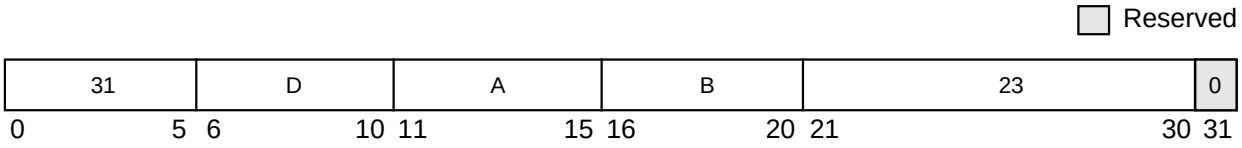
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

lwzx lwzx

Load Word and Zero Indexed (x'7C00 002E')

lwzx **r**D,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + rB
rD ← MEM(EA, 4)
```

EA is the sum (**r**A|0) + (**r**B). The word in memory addressed by EA is loaded into **r**D.

Other registers altered:

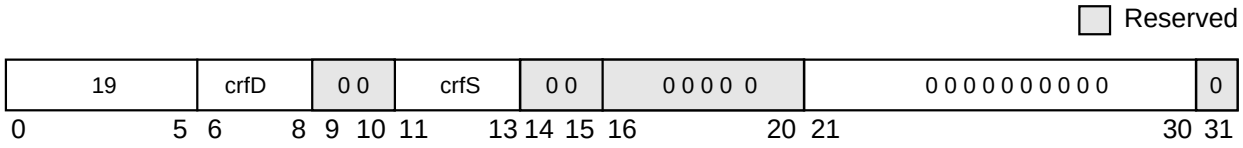
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

mcrf mcrf

Move Condition Register Field (x’4C00 0000’)

mcrf **crf**D**,crf**S




```
CR[4 * crfD–4 * crfD + 3] ← CR[4 * crfS–4 * crfS + 3]
```

The contents of condition register field **crf**S are copied into condition register field **crf**D. All other condition register fields remain unchanged.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):

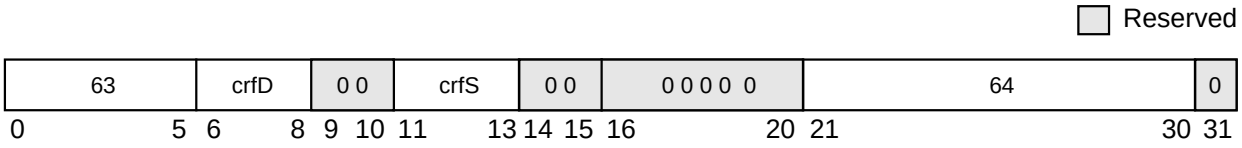
  Affected: LT, GT, EQ, SO

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XL

mcrfs mcrfs

Move to Condition Register from FPSCR (x'FC00 0080')

mcrfs **crf**D**,crf**S



The contents of FPSCR field **crf**S are copied to CR field **crf**D. All exception bits copied (except FEX and VX) are cleared in the FPSCR.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):

  Affected: FX, FEX, VX, OX
- Floating-Point Status and Control Register:

  Affected: FX, OX (if **crf**S = 0)

  Affected: UX, ZX, XX, VXSNAN (if **crf**S = 1)

  Affected: VXISI, VXIDI, VXZDZ, VXIMZ (if **crf**S = 2)

  Affected: VXVC (if **crf**S = 3)

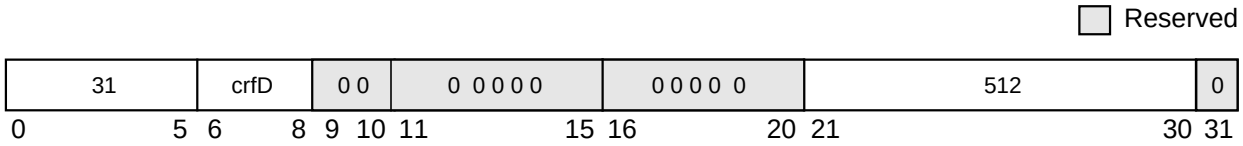
  Affected: VXSOFT, VXSQRT, VXCVI (if **crf**S = 5)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

mcrxr mcrxr

Move to Condition Register from XER (x'7C00 0400')

mcrxr **crf**D




```
CR[4 * crfD , 4 * crfD + 3] ← XER[0–3]
XER[0–3] ← 0b0000
```

The contents of XER[0–3] are copied into the condition register field designated by **crf**D. All other fields of the condition register remain unchanged. XER[0–3] is cleared.

Other registers altered:

- Condition Register (CR field specified by operand **crf**D):

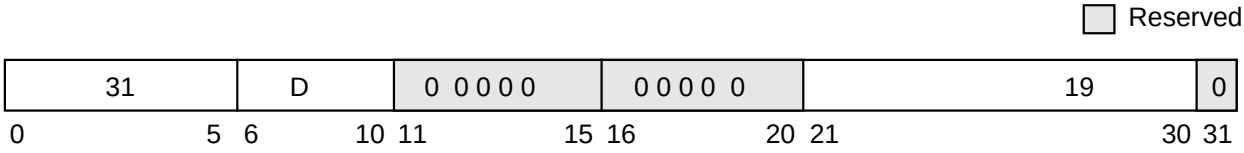
  Affected: LT, GT, EQ, SO
- XER[0–3]

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

mfcr mfcr

Move from Condition Register (x'7C00 0026')

mfcr **r**D



rD ← CR

The contents of the condition register (CR) are placed into **r**D.

Other registers altered:

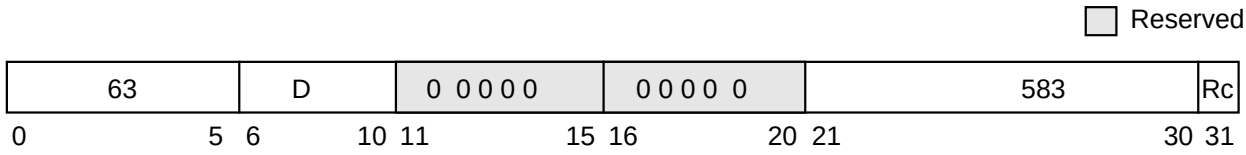
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

mffs*x* mffs*x*

Move from FPSCR (x'FC00 048E')

mffs **fr**D (Rc = 0)
mffs. **fr**D (Rc = 1)




```
frD[32-63] ← FPSCR
```

The contents of the floating-point status and control register (FPSCR) are placed into the low-order bits of register **fr**D. The high-order bits of register **fr**D are undefined.

Other registers altered:

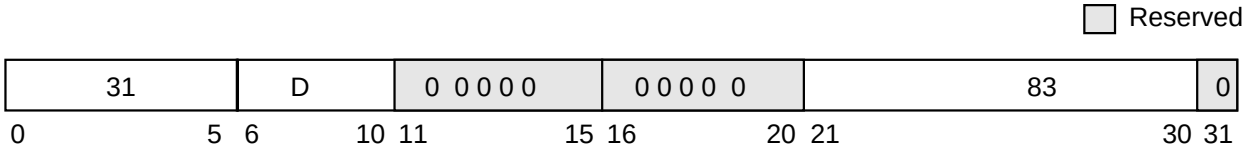
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

mfmsr mfmsr

Move from Machine State Register (x'7C00 00A6')

mfmsr **r**D



rD ← MSR

The contents of the MSR are placed into **r**D.

This is a supervisor-level instruction.

Other registers altered

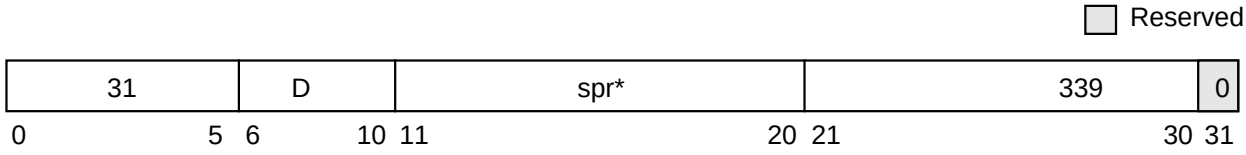
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
OEA	Yes			X

mfspr mfspr

Move from Special-Purpose Register (x'7C00 02A6')

mfspr **r**D**,**SPR



***Note:** This is a split field.

```
n ← spr[5–9] || spr[0–4]
rD ← SPR(n)
```

In the PowerPC UISA, the SPR field denotes a special-purpose register, encoded as shown in Table 12-9 .The contents of the designated special purpose register are placed into **r**D.

Table 12-9. Gekko UISA SPR Encodings for mfspr

SPR**			Register Name
Decimal	spr[5–9]	spr[0–4]	
1	00000	00001	XER
8	00000	01000	LR
9	00000	01001	CTR

** Note that the order of the two 5-bit halves of the SPR number is reversed compared with the actual instruction coding.

If the SPR field contains any value other than one of the values shown in Table 12-9 (and the processor is in user mode), one of the following occurs:

- The system illegal instruction error handler is invoked.
- The system supervisor-level instruction error handler is invoked.
- The results are boundedly undefined.

Other registers altered:

- None

Simplified mnemonics:

mfxerrD	equivalent to	**mfspr r**D**,1**
mflr rD	equivalent to	**mfspr r**D**,8**
mfctrrD	equivalent to	**mfspr r**D**,9**

In the PowerPC OEA, the SPR field denotes a special-purpose register, encoded as shown in Table 12-10. The contents of the designated SPR are placed into **r**D.

In the PowerPC UISA, the SPR field denotes a special-purpose register, encoded as shown in Table 12-10. If the SPR[0] = 0 (Access type User), the contents of the designated SPR are placed into **r**D.

SPR[0] = 1 if and only if reading the register is supervisor-level. Execution of this instruction specifying a defined and supervisor-level register when MSR[PR] = 1 will result in a privileged instruction type program exception.

If MSR[PR] = 1, the only effect of executing an instruction with an SPR number that is not shown in Table 12-10 and has SPR[0] = 1 is to cause a supervisor-level instruction type program exception or an illegal instruction type program exception. For all other cases, MSR[PR] = 0 or SPR[0] = 0.

If the SPR field contains any value that is not shown in Table 12-10, either an illegal instruction type program exception occurs or the results are boundedly undefined.

Table 12-10. Gekko OEA SPR Encodings for mfspr

SPR[1]			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
1	00000	00001	XER	User
8	00000	01000	LR	User
9	00000	01001	CTR	User
18	00000	10010	DSISR	Supervisor
19	00000	10011	DAR	Supervisor
22	00000	10110	DEC	Supervisor
25	00000	11001	SDR1	Supervisor
26	00000	11010	SRR0	Supervisor
27	00000	11011	SRR1	Supervisor
272	01000	10000	SPRG0	Supervisor
273	01000	10001	SPRG1	Supervisor
274	01000	10010	SPRG2	Supervisor
275	01000	10011	SPRG3	Supervisor
282	01000	11010	EAR	Supervisor
287	01000	11111	PVR	Supervisor
528	10000	10000	IBAT0U	Supervisor
529	10000	10001	IBAT0L	Supervisor
530	10000	10010	IBAT1U	Supervisor
531	10000	10011	IBAT1L	Supervisor
532	10000	10100	IBAT2U	Supervisor
533	10000	10101	IBAT2L	Supervisor
534	10000	10110	IBAT3U	Supervisor
535	10000	10111	IBAT3L	Supervisor

Table 12-10. Gekko OEA SPR Encodings for mfspr (Continued)

SPR[1]			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
536	10000	11000	DBAT0U	Supervisor
537	10000	11001	DBAT0L	Supervisor
538	10000	11010	DBAT1U	Supervisor
539	10000	11011	DBAT1L	Supervisor
540	10000	11100	DBAT2U	Supervisor
541	10000	11101	DBAT2L	Supervisor
542	10000	11110	DBAT3U	Supervisor
543	10000	11111	DBAT3L	Supervisor
912	11100	10000	GQR0	Supervisor
913	11100	10001	GQR1	Supervisor
914	11100	10010	GQR2	Supervisor
915	11100	10011	GQR3	Supervisor
916	11100	10100	GQR4	Supervisor
917	11100	10101	GQR5	Supervisor
918	11100	10110	GQR6	Supervisor
919	11100	10111	GQR7	Supervisor
920	11100	11000	HID2	Supervisor
921	11100	11001	WPAR	Supervisor
922	11100	11010	DMA_U	Supervisor
923	11100	11011	DMA_L	Supervisor
936	11101	01000	UMMCR0	User
937	11101	01001	UPMC1	User
938	11101	01010	UPMC2	User
939	11101	01011	USIA	User
940	11101	01100	UMMCR1	User
941	11101	01101	UPMC3	User
942	11101	01110	UPMC4	User
943	11101	01111	USDA	User
952	11101	11000	MMCR0	Supervisor
953	11101	11001	PMC1	Supervisor
954	11101	11010	PMC2	Supervisor
955	11101	11011	SIA	Supervisor
956	11101	11100	MMCR1	Supervisor
957	11101	11101	PMC3	Supervisor
958	11101	11110	PMC4	Supervisor
959	11101	11111	SDA	Supervisor
1008	11111	10000	HID0	Supervisor
1009	11111	10001	HID1	Supervisor
1010	11111	10010	IABR	Supervisor

Table 12-10. Gekko OEA SPR Encodings for mfspr (Continued)

SPR[1]			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
1013	11111	10101	DABR	Supervisor
1017	11111	11001	L2CR	Supervisor
1019	11111	11011	ICTC	Supervisor
1020	11111	11100	THRM1	Supervisor
1021	11111	11101	THRM2	Supervisor
1022	11111	11110	THRM3	Supervisor

[1]Note that the order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding.

For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order five bits appearing in bits 16–20 of the instruction and the low-order five bits in bits 11–15.

---

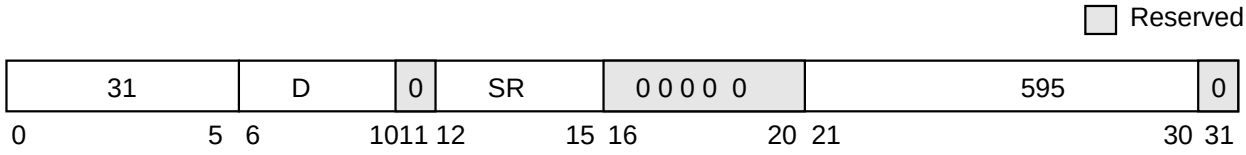
* Note that **mfspr** is supervisor-level only if SPR[0] = 1.

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA/OEA	Yes*			XFX

mfsr mfsr

Move from Segment Register (x'7C00 04A6')

mfsr rD,SR



rD ← SEGREG(**SR**)

The contents of the segment register SR are copied into **r**D.

This is a supervisor-level instruction.

Other registers altered:

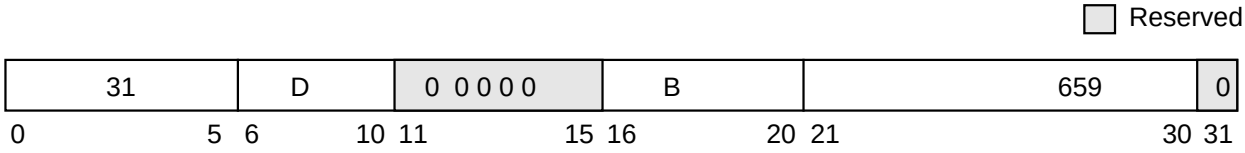
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
OEA	Yes			X

mfsrin mfsrin

Move from Segment Register Indirect (x'7C00 0526')

mfsrin **r**D,**r**B




```
rD ← SEGREG(rB[0–3])
```

The contents of the segment register selected by bits 0–3 of **r**B are copied into **r**D.

This is a supervisor-level instruction.

NOTE: The **r**A field is not defined for the **mfsrin** instruction in the PowerPC architecture. However, **mfsrin** performs the same function in the PowerPC architecture as does the **mfsri** instruction in the POWER architecture (if **r**A = 0).

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
OEA	Yes			X

mftb mftb

Move from Time Base (x’7C00 02E6’)

mftb **r**D,TBR

***Note:** This is a split field.

```
n ← tbr[5–9] || tbr[0–4]
if n = 268
then rD ← TBL
else if n = 269
        then rD ← TBU
        else error(invalid TBR field)
```

The contents of TBL or TBU are copied into rD, as designated by the value in TBR, encoded as shown here.

Table 12-11. TBR Encodings for mftb

TBR*			Register Name	Access
Decimal	tbr[5–9]	tbr[0–4]		
268	01000	01100	TBL	User
269	01000	01101	TBU	User

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

If the TBR field contains any value other than one of the values shown in Table 12-11, then one of the following occurs:

- The system illegal instruction error handler is invoked.
- The system supervisor-level instruction error handler is invoked.
- The results are boundedly undefined.

Important Note: Some implementations may implement **mftb** and **mfspr** identically, therefore, a TBR number should not match an SPR number.

For more information on the time base refer to Section 2.2, “PowerPC VEA Register Set—Time Base,” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Other registers altered:

- None

Simplified mnemonics:

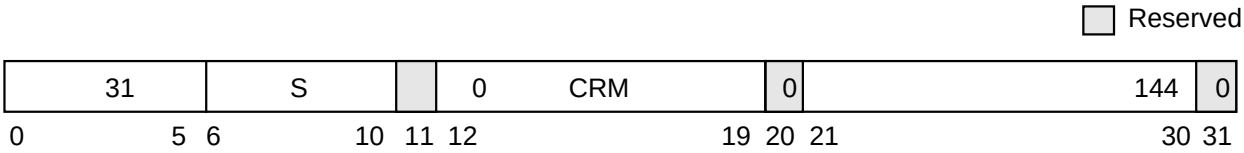
mftb rD	equivalent to	**mftb** r**D,268**
mftburD	equivalent to	**mftb** r**D,269**

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
VEA				XFX

mtcrf mtcrf

Move to Condition Register Fields (x'7C00 0120')

mtcrf CRM**,r**S




```
mask ← (4)(CRM[0]) || (4)(CRM[1]) ||... (4)(CRM[7])
CR ← (rS & mask) | (CR & ¬ mask)
```

The contents of **r**S are placed into the condition register under control of the field mask specified by CRM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0–7. If CRM(i) = 1, CR field i (CR bits 4 ∗ i through 4 ∗ i + 3) is set to the contents of the corresponding field of **r**S.

NOTE: Updating a subset of the eight fields of the condition register may have substantially poorer performance on some implementations than updating all of the fields.

Other registers altered:

- CR fields selected by mask

Simplified mnemonics:

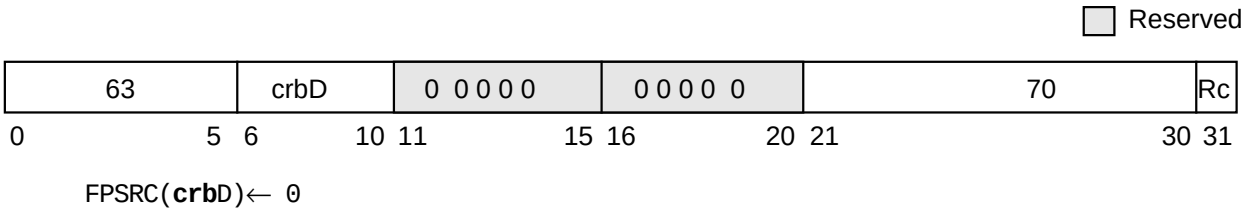
mtcr rS equivalent to **mtcrf** 0xFF**,r**S

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XFX

mtfsb0*x* mtfsb0*x*

Move to FPSCR Bit 0 (x'FC00 008C')

mtfsb0 **crb**D (Rc = 0)
mtfsb0. **crb**D (Rc = 1)




```
FPSRC(crbD)← 0
```

Bit **crb**D of the FPSCR is cleared.

Other registers altered:

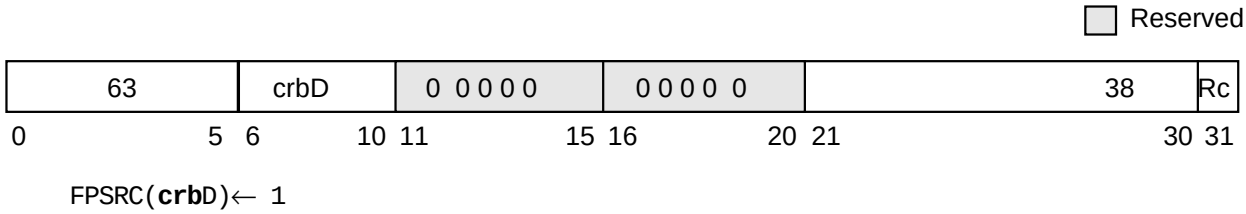
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX(if Rc = 1)
- Floating-Point Status and Control Register:
  Affected: FPSCR bit **crb**D
  NOTE: Bits 1 and 2 (FEX and VX) cannot be explicitly cleared.

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

mtfsb1*x* mtfsb1*x*

Move to FPSCR Bit 1 (x'FC00 004C')

mtfsb1 **crb**D (Rc = 0)
mtfsb1. **crb**D (Rc = 1)



FPSRC(**crb**D)← 1

Bit **crb**D of the FPSCR is set.

Other registers altered:

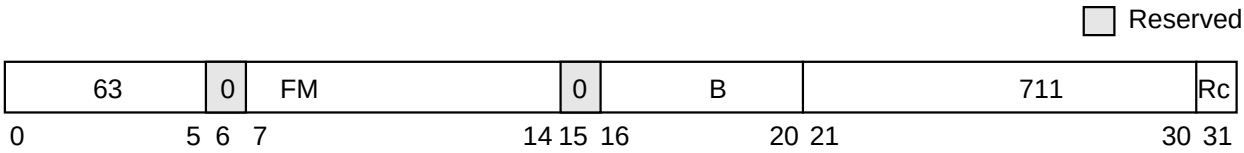
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX(if Rc = 1)
- Floating-Point Status and Control Register:
  Affected: FPSCR bit **crb**D and FX
  NOTE: Bits 1 and 2 (FEX and VX) cannot be explicitly set.

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

mtfsf*x* mtfsf*x*

Move to FPSCR Fields (x'FC00 058E')

mtfsf	FM,**fr**B	(Rc = 0)
mtfsf.	FM,**fr**B	(Rc = 1)



The low-order 32 bits of **fr**B are placed into the FPSCR under control of the field mask specified by FM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0–7. If FM[i] = 1, FPSCR field i (FPSCR bits 4 * i through 4 * i + 3) is set to the contents of the corresponding field of the low-order 32 bits of register **fr**B.

FPSCR[FX] is altered only if FM[0] = 1.

Updating fewer than all eight fields of the FPSCR may have substantially poorer performance on some implementations than updating all the fields.

When FPSCR[0–3] is specified, bits 0 (FX) and 3 (OX) are set to the values of **fr**B[32] and **fr**B[35] (that is, even if this instruction causes OX to change from 0 to 1, FX is set from **fr**B[32] and not by the usual rule that FX is set when an exception bit changes from 0 to 1). Bits 1 and 2 (FEX and VX) are set according to the usual rule and not from **fr**B[33–34].

Other registers altered:

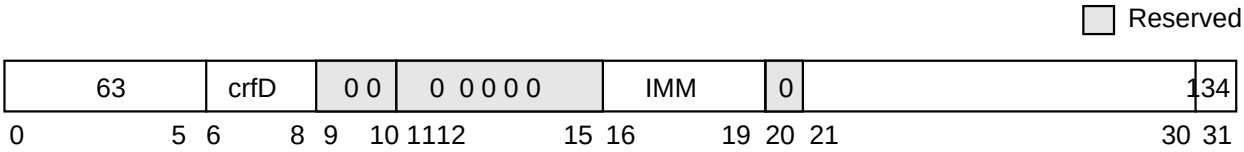
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX(if Rc = 1)
- Floating-Point Status and Control Register:
  Affected: FPSCR fields selected by mask

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XFL

mtfsfi*x* mtfsfi*x*

Move to FPSCR Field Immediate (x'FC00 010C')

mtfsfi **crf**D**,**IMM (Rc = 0)
mtfsfi. **crf**D**,**IMM (Rc = 1)



FPSCR[**crf**D] ← IMM

The value of the IMM field is placed into FPSCR field **crf**D.

FPSCR[FX] is altered only if **crf**D = 0.

When FPSCR[0–3] is specified, bits 0 (FX) and 3 (OX) are set to the values of IMM[0] and IMM[3] (that is, even if this instruction causes OX to change from 0 to 1, FX is set from IMM[0] and not by the usual rule that FX is set when an exception bit changes from 0 to 1). Bits 1 and 2 (FEX and VX) are set according to the usual rule and not from IMM[1–2].

Other registers altered:

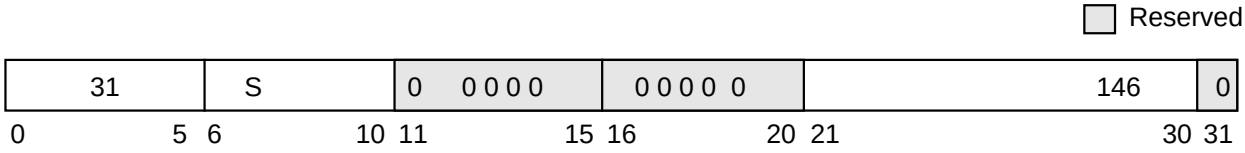
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX(if Rc = 1)
- Floating-Point Status and Control Register:
  Affected: FPSCR field **crf**D

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

mtmsr mtmsr

Move to Machine State Register (x'7C00 0124')

mtmsr **r**S



MSR ← (**r**S)

The contents of **r**S are placed into the MSR.

This is a supervisor-level instruction. It is also an execution synchronizing instruction except with respect to alterations to the POW and LE bits. Refer to Section 2.3.18, "Synchronization Requirements for Special Registers and for Lookaside Buffers" in the the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

In addition, alterations to the MSR[EE] and MSR[RI] bits are effective as soon as the instruction completes. Thus if MSR[EE] = 0 and an external or decrementer exception is pending, executing an **mtmsr** instruction that sets MSR[EE] = 1 will cause the external or decrementer exception to be taken before the next instruction is executed, if no higher priority exception exists.

Other registers altered:

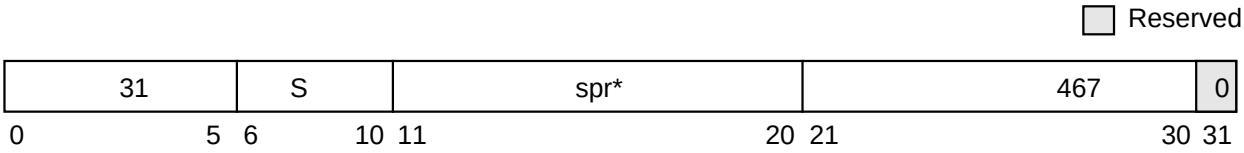
- MSR

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
OEA	Yes			X

mtspr mtspr

Move to Special-Purpose Register (x'7C00 03A6')

mtspr SPR**,rS**



***Note:** This is a split field.

```
n ← spr[5–9] || spr[0–4]
```

In the PowerPC UISA, the SPR field denotes a special-purpose register, encoded as shown in Table 12-12. The contents of **r**S are placed into the designated special-purpose register

Table 12-12. Gekko UISA SPR Encodings for mtspr

SPR**			Register Name
Decimal	spr[5–9]	spr[0–4]	
1	00000	00001	XER
8	00000	01000	LR
9	00000	01001	CTR

** Note that the order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding.

If the SPR field contains any value other than one of the values shown in Table 12-12, and the processor is operating in user mode, one of the following occurs:

- The system illegal instruction error handler is invoked.
- The system supervisor instruction error handler is invoked.
- The results are boundedly undefined.

Simplified mnemonics:

mtxerrD	equivalent to	**mtspr 1,r**D
mtlr rD	equivalent to	**mtspr 8,r**D
mtctrrD	equivalent to	**mtspr 9,r**D

In the PowerPC OEA, the SPR field denotes a special-purpose register, encoded as shown in Table 12-13. The contents of **r**S are placed into the designated special-purpose registerIn the PowerPC UISA, if the SPR[0]=0 (Access is User) the contents of **r**S are placed into the designated special-purpose register

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

The value of SPR[0] = 1 if and only if writing the register is a supervisor-level operation. Execution of this instruction specifying a defined and supervisor-level register when MSR[PR] = 1 results in a privileged instruction type program exception.

If MSR[PR] = 1 then the only effect of executing an instruction with an SPR number that is not shown in Table 12-13 and has SPR[0] = 1 is to cause a privileged instruction type program exception or an illegal instruction type program exception. For all other cases, MSR[PR] = 0 or SPR[0] = 0, if the SPR field contains any value that is not shown in Table 12-13, either an illegal instruction type program exception occurs or the results are boundedly undefined.

Other registers altered:

- See Table 12-13.

Table 12-13. Gekko OEA SPR Encodings for mtspr

SPR[1]			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
1	00000	00001	XER	User
8	00000	01000	LR	User
9	00000	01001	CTR	User
18	00000	10010	DSISR	Supervisor
19	00000	10011	DAR	Supervisor
22	00000	10110	DEC	Supervisor
25	00000	11001	SDR1	Supervisor
26	00000	11010	SRR0	Supervisor
27	00000	11011	SRR1	Supervisor
272	01000	10000	SPRG0	Supervisor
273	01000	10001	SPRG1	Supervisor
274	01000	10010	SPRG2	Supervisor
275	01000	10011	SPRG3	Supervisor
282	01000	11010	EAR	Supervisor
284	01000	11100	TBL	Supervisor
285	01000	11101	TBU	Supervisor
528	10000	10000	IBAT0U	Supervisor
529	10000	10001	IBAT0L	Supervisor
530	10000	10010	IBAT1U	Supervisor
531	10000	10011	IBAT1L	Supervisor
532	10000	10100	IBAT2U	Supervisor
533	10000	10101	IBAT2L	Supervisor
534	10000	10110	IBAT3U	Supervisor

Table 12-13. Gekko OEA SPR Encodings for mtspr (Continued)

SPR[1]			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
535	10000	10111	IBAT3L	Supervisor
536	10000	11000	DBAT0U	Supervisor
537	10000	11001	DBAT0L	Supervisor
538	10000	11010	DBAT1U	Supervisor
539	10000	11011	DBAT1L	Supervisor
540	10000	11100	DBAT2U	Supervisor
541	10000	11101	DBAT2L	Supervisor
542	10000	11110	DBAT3U	Supervisor
543	10000	11111	DBAT3L	Supervisor
912	11100	10000	GQR0	Supervisor
913	11100	10001	GQR1	Supervisor
914	11100	10010	GQR2	Supervisor
915	11100	10011	GQR3	Supervisor
916	11100	10100	GQR4	Supervisor
917	11100	10101	GQR5	Supervisor
918	11100	10110	GQR6	Supervisor
919	11100	10111	GQR7	Supervisor
920	11100	11000	HID2	Supervisor
921	11100	11001	WPAR	Supervisor
922	11100	11010	DMA_U	Supervisor
923	11100	11011	DMA_L	Supervisor
936	11101	01000	UMMCR0	User
937	11101	01001	UPMC1	User
938	11101	01010	UPMC2	User
939	11101	01011	USIA	User
940	11101	01100	UMMCR1	User
941	11101	01101	UPMC3	User
942	11101	01110	UPMC4	User
943	11101	01111	USDA	User
952	11101	11000	MMCR0	Supervisor
953	11101	11001	PMC1	Supervisor
954	11101	11010	PMC2	Supervisor
955	11101	11011	SIA	Supervisor
956	11101	11100	MMCR1	Supervisor
957	11101	11101	PMC3	Supervisor
958	11101	11110	PMC4	Supervisor
959	11101	11111	SDA	Supervisor
1008	11111	10000	HID0	Supervisor
1009	11111	10001	HID1	Supervisor

Table 12-13. Gekko OEA SPR Encodings for mtspr (Continued)

SPR[1]			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
1010	11111	10010	IABR	Supervisor
1013	11111	10101	DABR	Supervisor
1017	11111	11001	L2CR	Supervisor
1019	11111	11011	ICTC	Supervisor
1020	11111	11100	THRM1	Supervisor
1021	11111	11101	THRM2	Supervisor
1022	11111	11110	THRM3	Supervisor

[1]Note that the order of the two 5-bit halves of the SPR number is reversed. For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order five bits appearing in bits 16–20 of the instruction and the low-order five bits in bits 11–15.

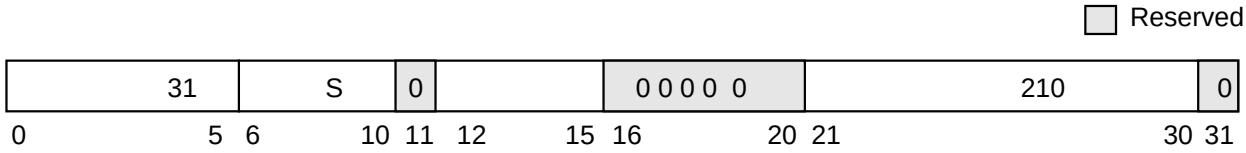
NOTE: **mfspr** is supervisor-level only if SPR[0] = 1.

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
OEA	Yes			XFX

mtsr mtsr

Move to Segment Register (x'7C00 01A4')

mtsr SR,**r**S



SEGREG(SR) ← (**r**S)

The contents of **r**S are placed into SR.

This is a supervisor-level instruction.

Other registers altered:

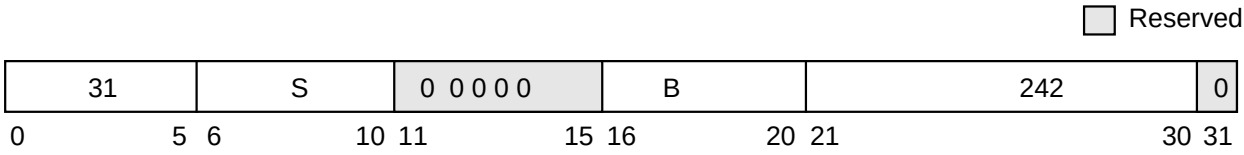
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
OEA	Yes			X

mtsrin mtsrin

Move to Segment Register Indirect (x'7C00 01E4')

mtsrin **r**S,**r**B




```
SEGREG(rB[0–3]) ← (rS)
```

The contents of **r**S are copied to the segment register selected by bits 0–3 of **r**B.

This is a supervisor-level instruction.

NOTE: The PowerPC architecture does not define the **r**A field for the **mtsrin** instruction. However, **mtsrin** performs the same function in the PowerPC architecture as does the **mtsri** instruction in the POWER architecture (if **r**A = 0).

Other registers altered:

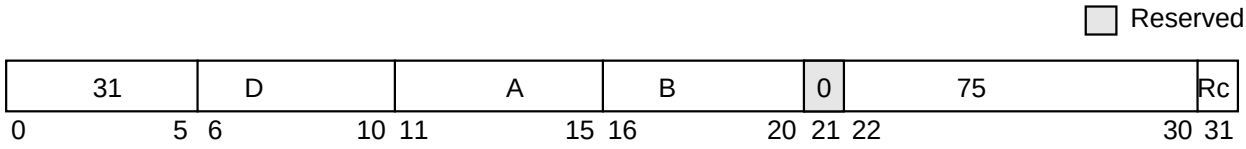
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
OEA	Yes			X

mulhwx mulhwx

Multiply High Word (x'7C00 0096')

mulhw **r**D,**r**A,**r**B (Rc = 0)
mulhw. **r**D,**r**A,**r**B (Rc = 1)




```
prod[0–63] ← (rA * (rB
rD ← prod
```

The 32-bit product is formed from the contents **r**A and **r**B. The high-order 32 bits of the 64-bit product of the operands are placed into **r**D. Both the operands and the product are interpreted as signed integers.

This instruction may execute faster on some implementations if **r**B contains the operand having the smaller absolute value.

Other registers altered:

- Condition Register (CR0 field):

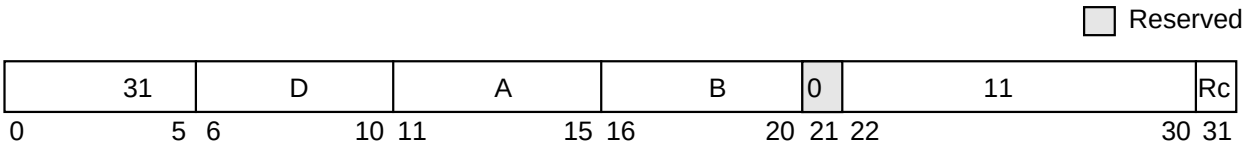
  Affected: LT, GT, EQ, SO (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

mulhwu*x* mulhwu*x*

Multiply High Word Unsigned (x'7C00 0016')

mulhwu **r**D,**r**A,**r**B (Rc = 0)
mulhwu. **r**D,**r**A,**r**B (Rc = 1)




```
prod[0–63] ← (rA) * (rB
rD ← prod[0–31]
```

The 32-bit operands are the contents **r**A and **r**B. The high-order 32 bits of the 64-bit product of the operands are placed into **r**D.

Both the operands and the product are interpreted as unsigned integers, except that if Rc = 1 the first three bits of CR0 field are set by signed comparison of the result to zero.

This instruction may execute faster on some implementations if **r**B contains the operand having the smaller absolute value.

Other registers altered:

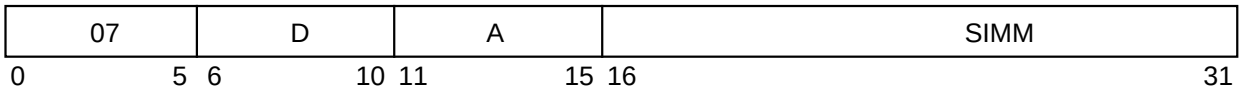
- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO(if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

mulli mulli

Multiply Low Immediate (x'1C00 0000')

mulli **r**D,**r**A,SIMM




```
prod[0–48] ← (rA) * SIMM
rD ← prod[16-48]
```

The first operand is **r**A. The second operand is the value of the SIMM field. The low-order 32-bits of the 48-bit product of the operands are placed into **r**D.

Both the operands and the product are interpreted as signed integers. The low-order of the product are calculated independently of whether the operands are treated as signed or unsigned 32-bit integers.

This instruction can be used with **mulhd***x* or **mulhw***x* to calculate a full 64-bit product.

Other registers altered:

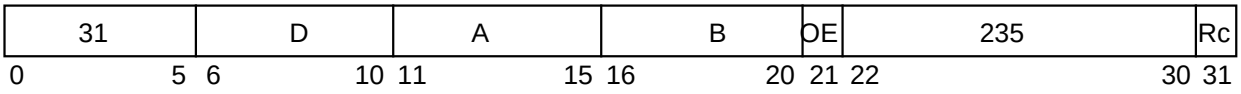
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

mullw*x* mullw*x*

Multiply Low Word (x'7C00 01D6')

mullw	**r**D,**r**A,**r**B	(OE = 0 Rc = 0)
mullw.	**r**D,**r**A,**r**B	(OE = 0 Rc = 1)
mullwo	**r**D,**r**A,**r**B	(OE = 1 Rc = 0)
mullwo.	**r**D,**r**A,**r**B	(OE = 1 Rc = 1)




```
prod[0–48] ← (rA) * (rB
rD ← prod[16-48]
```

The 32-bit operands are the contents of **r**A and **r**B. The low-order of the 64-bit product (**r**A) * **(r**B) are placed into **r**D.

The low-order 32-bits of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

If OE = 1, then OV is set if the product cannot be represented in 32 bits. Both the operands and the product are interpreted as signed integers.

NOTE: This instruction may execute faster on some implementations if **r**B contains the operand having the smaller absolute value.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO(if Rc = 1)

  NOTE: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).

- XER:

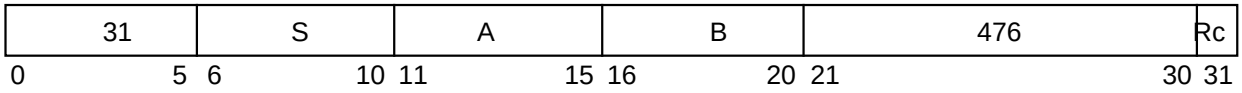
  Affected: SO, OV (if OE = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

nand*x* nand*x*

NAND (x’7C00 03B8’)

nand **r**A**,r**S**,r**B (Rc = 0)
nand. **r**A**,r**S**,r**B (Rc = 1)



rA ← ¬ ((**r**S) & (**r**B))

The contents of **r**S are ANDed with the contents of **r**B and the complemented result is placed into **r**A.

nand with **r**S = **r**B can be used to obtain the one's complement.

Other registers altered:

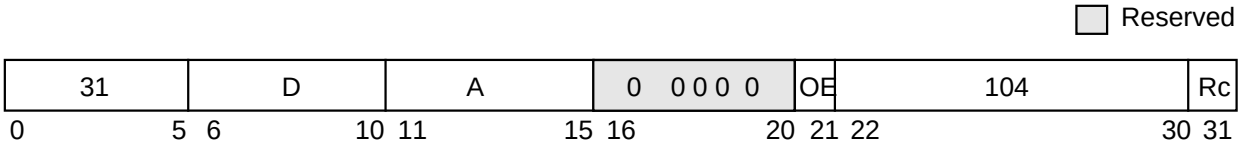
- Condition Register (CR0 field):

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

negx negx

Negate (x'7C00 00D0')

neg	**r**D**,r**A	(OE = 0 Rc = 0)
neg.	**r**D**,r**A	(OE = 0 Rc = 1)
nego	**r**D**,r**A	(OE = 1 Rc = 0)
nego.	**r**D**,r**A	(OE = 1 Rc = 1)



rD ← ¬ (rA) + 1

The value 1 is added to the complement of the value in **r**A, and the resulting two's complement is placed into **r**D.

If **r**A contains the most negative 32-bit number (0x8000_0000), the reseult is the most negative number and if OE = 1, OV is set.

Other registers altered:

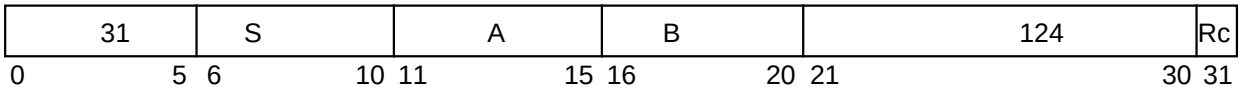
- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO(if Rc = 1)
- XER:
  Affected: SO OV (if OE = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

nor*x* nor*x*

NOR (x’7C00 00F8’)

nor **r**A,**r**S,**r**B (Rc = 0)
nor. **r**A,**r**S,**r**B (Rc = 1)



rA ← ¬ ((**r**S) | (**r**B))

The contents of **r**S are ORed with the contents of **r**B and the complemented result is placed into **r**A.

nor with **r**S = **r**B can be used to obtain the one’s complement.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO(if Rc = 1)

Simplified mnemonics:

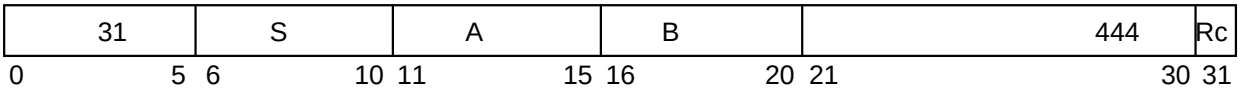
not **r**D,**r**S equivalent to **nor** **r**A,**r**S,**r**S

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

orx orx

OR (x'7C00 0378')

or **r**A,**r**S,**r**B (Rc = 0)
or. **r**A,**r**S,**r**B (Rc = 1)



rA ← (**r**S) | (**r**B)

The contents of **r**S are ORed with the contents of **r**B and the result is placed into **r**A.

The example under simplified mnemonic **mr** demonstrates the use of the **or** instruction to move register contents.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO(if Rc = 1)

Simplified mnemonics:

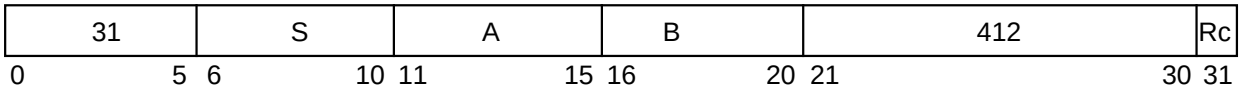
mr rA,rS equivalent to or rA,rS,rS

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

orc*x* orc*x*

OR with Complement (x’7C00 0338’)

orc **r**A,**r**S,**r**B (Rc = 0)
orc. **r**A,**r**S,**r**B (Rc = 1)



rA ← (**r**S) | ¬ (**r**B)

The contents of **r**S are ORed with the complement of the contents of **r**B and the result is placed into **r**A.

Other registers altered:

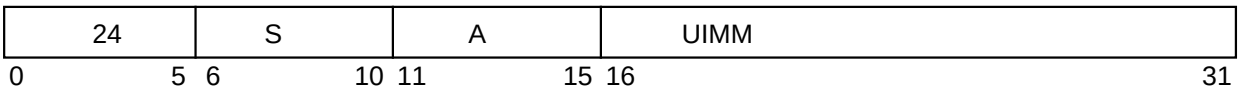
- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

ori ori

OR Immediate (x'6000 0000')

ori **r**A**,r**S**,**UIMM




```
rA ← (rS) | ((16)0 || UIMM)
```

The contents of **r**S are ORed with 0x0000 || UIMM and the result is placed into **r**A.

The preferred no-op (an instruction that does nothing) is **ori 0,0,0**.

Other registers altered:

- None

Simplified mnemonics:

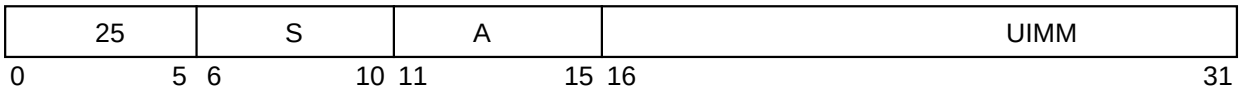
nop	equivalent to	**ori**	**0,0,0**

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

oris oris

OR Immediate Shifted (x'6400 0000')

oris **r**A,**r**S,UIMM



rA ← (**r**S) | (UIMM || (16)0)

The contents of **r**S are ORed with UIMM || 0x0000 and the result is placed into **r**A.

Other registers altered:

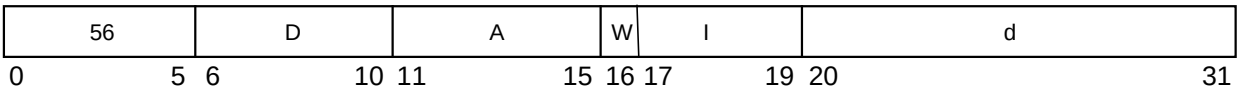
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

psq_l psq_l

Paired Single Quantized Load, (x'E000 0000')

psq_l f**r**D**,d(r**A**),W,I**




```
if HID2[PSE] = 0 | HID2[LSQE] = 0 then Goto illegal instruction error handler
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
lt ← qrI[LD_TYPE]
ls ← qrI[LD_SCALE]
c ← 4
if lt = (4|6) then c ← 10
if lt = (5|7) then c ← 20
if W = 0
then
     frD(ps0) ← dequantized(MEM(EA,c),lt,ls)
     frD(ps1) ← dequantized(MEM(EA+c,c),lt,ls)
else
     frD(ps0) ← dequantized(MEM(EA,c),lt,ls)
     frD(ps1) ← 1.0
```

PS0 and PS1 in **fr**D are loaded with a pair of single precision floating point numbers.

Memory is accessed at the effective address (EA is the sum (**r**A|0) + d) as defined by the instruction. A pair of numbers from memory are converted as defined by the indicated GQR control registers and the results are placed into PS0 and PS1. However, if W=1 then only one number is accessed from memory, converted according to GQR and placed into PS0. PS1 is loaded with a floating point value of 1.0.

The 3 bit field I selects one of the eight 32 bit GQR control registers. From this register the LOAD_SCALE and the LD_TYPE fields are used. The LD_TYPE field defines whether the data in memory is floating point or integer format. If the latter it also defines whether each integer is 8-bits or 16-bits, signed or unsigned. The LOAD_SCALE field is applied only to integer numbers and is a signed integer that is subtracted from the exponent after the integer number from memory has been converted to floating point format.
(See Section 2.3.4.3.12 for dequantized operation**.)**

Other registers altered:

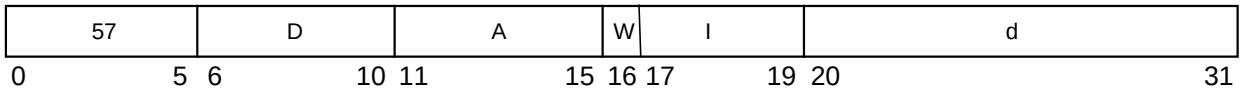
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		DW

psq_lu psq_lu

Paired Single Quantized Load with Update, (x'E400 0000')

psq_lu f**r**D**,d(r**A**),W,I**




```
if HID2[PSE] = 0 | HID2[LSQE] = 0 then Goto illegal instruction error handler
EA ← (rA) + EXTS(d)
lt ← qrI[LD_TYPE]
ls ← qrI[LD_SCALE]
c ← 4
if lt = (4|6) then c ← 10
if lt = (5|7) then c ← 20
if W = 0
then
      frD(ps0) ← dequantized(MEM(EA,c),lt,ls)
      frD(ps1) ← dequantized(MEM(EA+c,c),lt,ls)
else
      frD(ps0) ← dequantized(MEM(EA,c),lt,ls)
      frD(ps1) ← 1.0
rA ← EA
```

PS0 and PS1 in **fr**D are loaded with a pair of single-precision floating-point numbers.

Memory is accessed at the effective address (EA is the sum (**r**A) + d) as defined by the instruction. A pair of numbers from memory are converted as defined by the indicated GQR control registers and the results are placed into PS0 and PS1. However, if W=1 then only one number is accessed from memory, converted according to GQR and placed into PS0. PS1 is loaded with a floating point value of 1.0.

The 3 bit field I selects one of the eight 32 bit GQR control registers. From this register the LOAD_SCALE and the LD_TYPE fields are used. The LD_TYPE field defines whether the data in memory is floating point or integer format. If the latter it also defines whether each integer is 8-bits or 16-bits, signed or unsigned. The LOAD_SCALE field is applied only to integer numbers and is a signed integer that is subtracted from the exponent after the integer number from memory has been converted to floating point format.

(See Section 2.3.4.3.12 for dequantized operation**.)**

The effective address is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

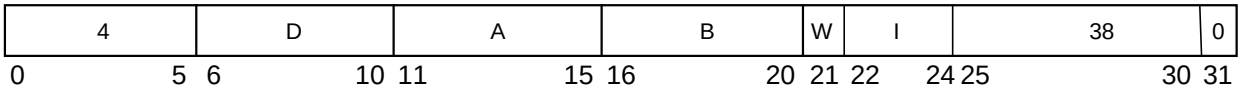
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		DW

psq_lux psq_lux

Paired Single Quantized Load with update Indexed, (x'1000 004C')

psq_lux f**r**D**, r**A,rB**,W,I**




```
if (HID2[PSE] = 0) then Goto illegal instruction error handler
EA ← (rA) + (rB)
lt ← qrI[LD_TYPE]
ls ← qrI[LD_SCALE]
c ← 4
if lt = (4|6) then c ← 10
if lt = (5|7) then c ← 20
if W = 0
then
     frD(ps0) ← dequantized(MEM(EA,c),lt,ls)
     frD(ps1) ← dequantized(MEM(EA+c,c),lt,ls)
else
     frD(ps0) ← dequantized(MEM(EA,c),lt,ls)
     frD(ps1) ← 1.0
rA←EA
```

PS0 and PS1 in frD are loaded with a pair of single precision floating point numbers.

Memory is accessed at the effective address (EA is the sum (**r**A) + (**r**B)) as defined by the instruction. A pair of numbers from memory are converted as defined by the indicated GQR control registers and the results are placed into PS0 and PS1. However, if W=1 then only one number is accessed from memory, converted according to GQR and placed into PS0. PS1 is loaded with a floating point value of 1.0.

The 3 bit field I selects one of the eight 32 bit GQR control registers. From this register the LOAD_SCALE and the LD_TYPE fields are used. The LD_TYPE field defines whether the data in memory is floating point or integer format. If the latter it also defines whether each integer is 8-bits or 16-bits, signed or unsigned. The LOAD_SCALE field is applied only to integer numbers and is a signed integer that is subtracted from the exponent after the integer number from memory has been converted to floating point format.
(See Section 2.3.4.3.12 for dequantized operation**.)**

The effective address is placed into register rA.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

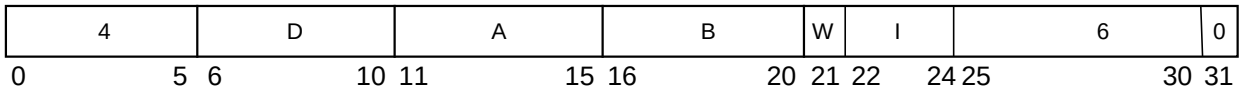
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		XW

psq_lx psq_lx

Paired Single Quantized Load Indexed, (x'1000 000C')

psq_lx f**r**D**, r**A,rB**,W,I**




```
if HID2[PSE] = 0 then Goto illegal instruction error handler
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
lt ← qrI[LD_TYPE]
ls ← qrI[LD_SCALE]
c ← 4
if lt = (4|6) then c ← 10
if lt = (5|7) then c ← 20
if W = 0
then
     frD(ps0) ← dequantized(MEM(EA,c),lt,ls)
     frD(ps1) ← dequantized(MEM(EA+c,c),lt,ls)
else
     frD(ps0) ← dequantized(MEM(EA,c),lt,ls)
     frD(ps1) ← 1.0
```

PS0 and PS1 in **fr**D are loaded with a pair of single precision floating point numbers.

Memory is accessed at the effective address (EA is the sum (**r**A|0) + (**r**B)) as defined by the instruction. A pair of numbers from memory are converted as defined by the indicated GQR control registers and the results are placed into PS0 and PS1. However, if W=1 then only one number is accessed from memory, converted according to GQR and placed into PS0. PS1 is loaded with a floating point value of 1.0.

The 3 bit field I selects one of the eight 32 bit GQR control registers. From this register the LOAD_SCALE and the LD_TYPE fields are used. The LD_TYPE field defines whether the data in memory is floating point or integer format. If the latter it also defines whether each integer is 8-bits or 16-bits, signed or unsigned. The LOAD_SCALE field is applied only to integer numbers and is a signed integer that is subtracted from the exponent after the integer number from memory has been converted to floating point format.
(See Section 2.3.4.3.12 for dequantized operation**.)**

Other registers altered:

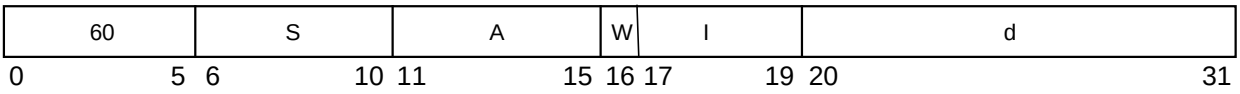
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		XW

psq_st psq_st

Paired Single Quantized Store, (x'F000 0000')

psq_st fr**S,d(r**A**),W,I**




```
if HID2[PSE] = 0 | HID2[LSQE] = 0 then Goto illegal instruction error handler
if rA = 0
then b← 0
else b ← (rA)
EA ← b + EXTS(d)
stt ← qrI[ST_TYPE]
sts ← qrI[ST_SCALE]
c ← 4
if stt = (4|6) then c ← 10
if stt = (5|7) then c ← 20
if W = 0
then
      MEM(EA,c) ← quantized(frS(ps0),stt,sts)
      MEM(EA+c,c) ← quantized(frS(ps1),stt,sts)
else
      MEM(EA,c) ← quantized(frS(ps0),stt,sts)
```

The effective address is the sum of (**r**A|0) + d as defined by the instruction. If W=1 only one floating point number from **fr**S(ps0) is quantized and stored to memory starting at the effective address. If W=0 a pair of floating point numbers from **fr**S(ps0) and **fr**S(ps1) are quantized and stored to memory starting at the effective address.

The 3 bit field I selects one of the eight 32 bit GQR control registers. From this register the STORE_SCALE and the ST_TYPE fields are used. The ST_TYPE field defines whether the data stored to memory is to be floating-point or integer format. If the latter it also defines whether each integer is 8-bits or 16-bits, signed or unsigned. The STORE_SCALE field is a signed integer that is added to the exponent of the floating point number before it is converted to integer and stored to memory.
(See Section 2.3.4.3.12 for dequantized operation**.)**

For floating point numbers stored to memory the addition of the STORE_SCALE field to the exponent does not take place.

Other registers altered:

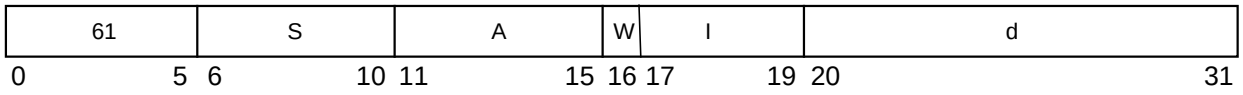
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		DW

psq_stu psq_stu

Paired Single Quantized Store with update, (x' F400 0000')

psq_stu f**r**S**,d(r**A**),W,I**




```
if HID2[PSE] = 0 | HID2[LSQE] = 0 then Goto illegal instruction error handler
EA ← (rA) + EXTS(d)
stt ← qrI[ST_TYPE]
sts ← qrI[ST_SCALE]
c ← 4
if stt = (4|6) then c ← 10
if stt = (5|7) then c ← 20
if W = 0
then
     MEM(EA,c) ← quantized(frS(ps0),stt,sts)
     MEM(EA+c,c) ← quantized(frS(ps1),stt,sts)
else
     MEM(EA,c) ← quantized(frS(ps0),stt,sts)
rA ← EA
```

The effective address is the sum of (**r**A) + d as defined by the instruction. If W=1 only one floating point number from **fr**S(ps0) is quantized and stored to memory starting at the effective address. If W=0 a pair of floating point numbers from **fr**S(ps0) and **fr**S(ps1) are quantized and stored to memory starting at the effective address.

The 3 bit field I selects one of the eight 32 bit GQR control registers. From this register the STORE_SCALE and the ST_TYPE fields are used. The ST_TYPE field defines whether the data stored to memory is to be floating-point or integer format. If the latter it also defines whether each integer is 8-bits or 16-bits, signed or unsigned. The STORE_SCALE field is a signed integer that is added to the exponent of the floating point number before it is converted to integer and stored to memory.

For floating point numbers stored to memory the addition of the STORE_SCALE field to the exponent field does not take place. (See Section 2.3.4.3.12 for dequantized operation**.)**

The effective address is placed into register rA.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

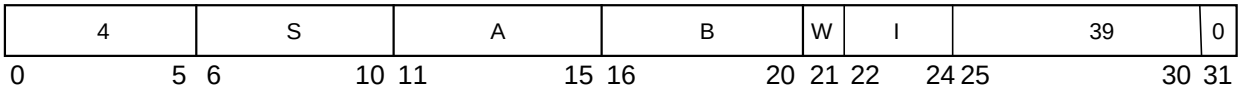
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		DW

psq_stux psq_stux

Paired Single Quantized Store with update Indexed, (x'1000 004E')

psq_stux f**r**S**, r**A,rB**,W,I**




```
if HID2[PSE] = 0 then Goto illegal instruction error handler
EA ← (rA) + (rB)
stt ← qrI[ST_TYPE]
sts ← qrI[ST_SCALE]
c ← 4
if stt = (4|6) then c ← 10
if stt = (5|7) then c ← 20
if W = 0
then
      MEM(EA,c) ← quantized(frS(ps0),stt,sts)
      MEM(EA+c,c) ← quantized(frS(ps1),stt,sts)
else
      MEM(EA,c) ← quantized(frS(ps0),stt,sts)
rA ← EA
```

The effective address is the sum of (**r**A) + (**r**B) as defined by the instruction. If W=1 only one floating point number from **fr**S(ps0) is quantized and stored to memory starting at the effective address. If W=0 a pair of floating point numbers from **fr**S(ps0) and **fr**S(ps1) are quantized and stored to memory starting at the effective address.

The 3 bit field I selects one of the eight 32 bit GQR control registers. From this register the STORE_SCALE and the ST_TYPE fields are used. The ST_TYPE field defines whether the data stored to memory is to be floating-point or integer format. If the latter it also defines whether each integer is 8-bits or 16-bits, signed or unsigned. The STORE_SCALE field is a signed integer that is added to the exponent of the floating point number before it is converted to integer and stored to memory.
(See Section 2.3.4.3.12 for dequantized operation**.)**

For floating point numbers stored to memory the addition of the STORE_SCALE field to the exponent field does not take place.

The effective address is placed into **r**A.
If **r**A = 0, the instruction form is invalid.

Other registers altered:

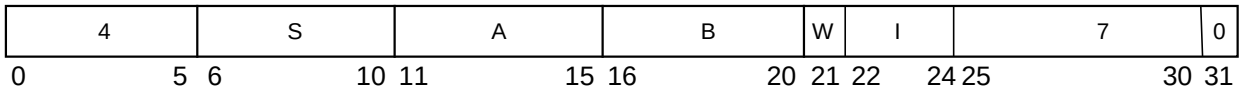
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		XW

psq_stx psq_stx

Paired Single Quantized Store Indexed, (x'1000 000E')

psq_stx f**r**S**, r**A,rB**,W,I**




```
if HID2[PSE] = 0 then Goto illegal instruction error handler
if rA = 0
then b ← 0
else b← (rA)
EA ← b + (rB)
stt ← qrI[ST_TYPE]
sts ← qrI[ST_SCALE]
c ← 4
if stt = (4|6) then c ← 10
if stt = (5|7) then c ← 20
if W = 0
then
      MEM(EA,c) ← quantized(frS(ps0),stt,sts)
      MEM(EA+c,c) ← quantized(frS(ps1),stt,sts)
else
      MEM(EA,c) ← quantized(frS(ps0),stt,sts)
```

The effective address is the sum of (**r**A|0) + (**r**B) as defined by the instruction. If W=1 only one floating point number from **fr**S(ps0) is quantized and stored to memory starting at the effective address. If W=0 a pair of floating point numbers from **fr**S(ps0) and **fr**S(ps1) are quantized and stored to memory starting at the effective address.

The 3 bit field I selects one of the eight 32 bit GQR control registers. From this register the STORE_SCALE and the ST_TYPE fields are used. The ST_TYPE field defines whether the data stored to memory is to be floating-point or integer format. If the latter it also defines whether each integer is 8-bits or 16-bits, signed or unsigned. The STORE_SCALE field is a signed integer that is added to the exponent of the floating point number before it is converted to integer and stored to memory.
(See Section 2.3.4.3.12 for dequantized operation**.)**

For floating point numbers stored to memory the addition of the STORE_SCALE field to the exponent field does not take place.

Other registers altered:

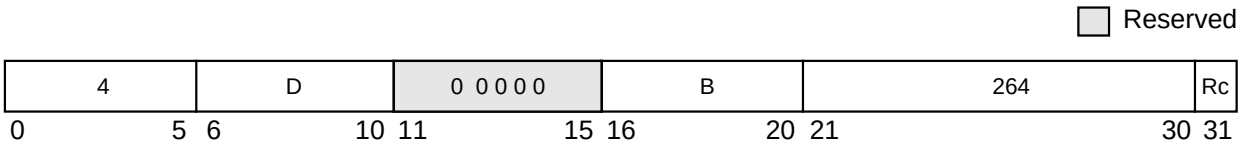
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		XW

ps_abs*x* ps_abs*x*

Paired Single Absolute Value (x’1000 0210’)

ps_abs	**fr**D,**fr**B	(Rc = 0)
ps_abs.	**fr**D,**fr**B	(Rc = 1)



The following operations are performed:

```
If HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← b’0’ || frB(ps0)[1-31]
frD(PS1) ← b’0’ || frB(ps1)[1-31]
```

The contents of **fr**B(ps0) with bit 0 cleared are placed into **fr**D(ps0).

The contents of **fr**B(ps1) with bit 0 cleared are placed into **fr**D(ps1).

Note that the **ps_abs** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN may be altered by **ps_abs**. This instruction does not alter the FPSCR.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

- Condition Register (CR1 field):

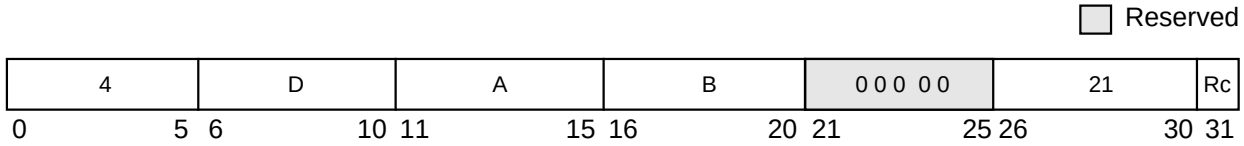
  Affected: FX, FEX, VX, OX(if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		X

ps_add*x* ps_add*x*

Paired Single Add (x'1000 002A')

ps_add **fr**D**,fr**A**,fr**B (Rc = 0)
ps_add. **fr**D**,fr**A**,fr**B (Rc = 1)



The following operations are performed:

```
If HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← frA(ps0) + frB(ps0)
frD(ps1) ← frA(ps1) + frB(ps1)
```

The floating-point operand in **fr**A(ps0) is added to the floating-point operand in **fr**B(ps0). If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps0).

The floating-point operand in **fr**A(ps1) is added to the floating-point operand in **fr**B(ps1). If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps1).

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands. All 25 bits in the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one. FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

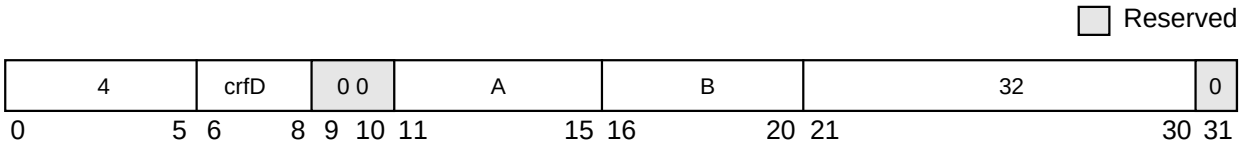
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control register(FPSCR):
  Affected: FPRF(ps0 only), FR, FI, FX, OX, UX, XX,VXSNAN, VXISI

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_cmpo0 ps_cmpo0

Paired Singles Compare Ordered High (x'1000 0040')

ps_cmpo0 **crf**D**,fr**A**,fr**B




```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
if (frA(ps0) is a NaN or (frB(ps0) is a NaN )
then c ← 0b0001
else if (frA(ps0)< frB(ps0))
        then c ← 0b1000
        else if (frA(ps0)> frB(ps0))
             then c ← 0b0100
             else c ← 0b0010
FPCC ← c
CR[(4 * crfD),(4 * crfD + 3)] ← c

if (frA(ps0) is an SNaN or frB(ps0) is an SNaN)
then
        VXSNAN ← 1
        if VE = 0 then VXVC ← 1
else if (frA(ps0) is a QNaN or frB(ps0) is a QNaN )
        then VXVC ← 1
```

The floating-point operand in **fr**A(ps0) is compared to the floating-point operand in **fr**B(ps0). The result of the compare is placed into CR field **crf**D and the FPCC.

If one of the operands is a NaN, either quiet or signaling, then CR field **crf**D and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, then VXSNAN is set, and if invalid operation is disabled (VE = 0) then VXVC is set. Otherwise, if one of the operands is a QNaN, then VXVC is set.

Other registers altered: (exception conditions are based on ps0 values)

- Condition Register (CR field specified by operand **crf**D):

  Affected: LT, GT, EQ, UN

- Floating-Point Status and Control Register:

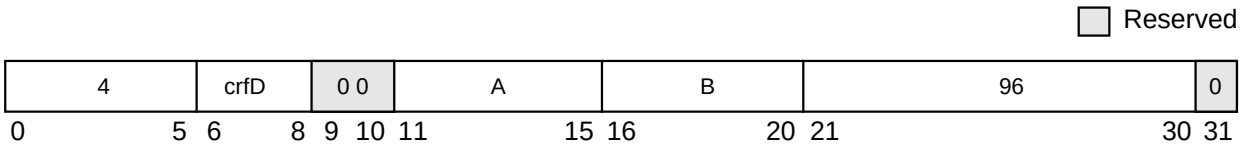
  Affected: FPCC(ps0 only), FX, VXSNAN, VXVC

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		X

ps_cmpo1 ps_cmpo1

Paired Singles Compare Ordered Low (x'1000 00C0')

ps_cmpol **crf**D**,fr**A**,fr**B




```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
if (frA(ps1) is a NaN or frB(ps1) is a NaN )
then c ← 0b0001
else if (frA(ps1)< frB(ps1))
        then c ← 0b1000
        else if (frA(ps1)> frB(ps1))
              then c ← 0b0100
              else c ← 0b0010
FPCC ← c
CR[(4 * crfD),(4 * crfD + 3)] ← c

if (frA(ps1) is an SNaN or frB(ps1) is an SNaN)
then
     VXSNAN ← 1
     if VE = 0 then VXVC ← 1
else if (frA(ps1)) is a QNaN or frB(ps1)) is a QNaN)
        then VXVC ← 1
```

The floating-point operand in **fr**A(ps1) is compared to the floating-point operand in **fr**B(ps1). The result of the compare is placed into CR field **crf**D and the FPCC.

If one of the operands is a NaN, either quiet or signaling, then CR field **crf**D and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, then VXSNAN is set, and if invalid operation is disabled (VE = 0) then VXVC is set. Otherwise, if one of the operands is a QNaN, then VXVC is set.

Other registers altered: (exception conditions are based on ps1 values)

- Condition Register (CR field specified by operand **crf**D):

  Affected: LT, GT, EQ, UN

- Floating-Point Status and Control Register:

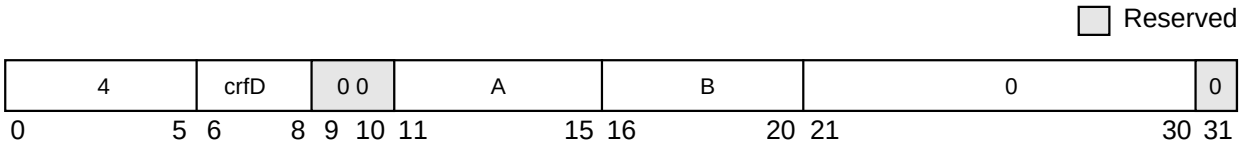
  Affected: FPCC(ps1 only), FX, VXSNAN, VXVC

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		X

ps_cmpu0 ps_cmpu0

Paired Singles Compare Unordered High (x'1000 0000')

ps_cmpu0 **crf**D**,fr**A**,fr**B




```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
if ( frA(ps0) is a NaN or frB(ps0) is a NaN )
then c ← 0b0001
else if (frA(ps0)< frB(ps0))
        then c ← 0b1000
        else if (frA(ps0)> frB(ps0))
              then c ← 0b0100
              else c ← 0b0010

FPCC ← c
CR[(4 * crfD),(4 * crfD + 3)] ← c

if (frA(ps0)) is an SNaN or (frB(ps0)) is an SNaN )
then VXSNAN ← 1
```

The floating-point operand in **fr**A(ps0) is compared to the floating-point operand in **fr**B(ps0). The result of the compare is placed into CR field **crf**D and the FPCC

If one of the operands is a NaN, either quiet or signaling, then CR field **crf**D and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, then VXSNAN is set.

Other registers altered: (exception conditions are based on ps0 values)

- Condition Register (CR field specified by operand **crf**D):

  Affected: LT, GT, EQ, UN

- Floating-Point Status and Control Register:

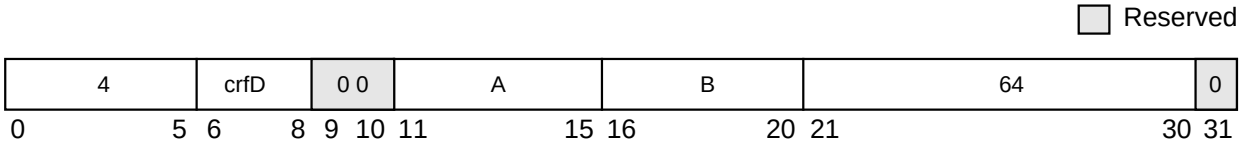
  Affected: FPCC(ps0 only), FX, VXSNAN

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		X

ps_cmpu1 ps_cmpu1

Paired Singles Compare Unordered Low(x'1000 0080')

ps_cmpul **crf**D**,fr**A**,fr**B




```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
if (frA(ps1) is a NaN or frB(ps1) is a NaN)
 then c ← 0b0001
 else if (frA(ps1)< frB(ps1))
        then c ← 0b1000
        else if (frA(ps1)> (frB(ps1))
           then c ← 0b0100
           else c ← 0b0010
FPCC ← c
CR[(4 * crfD),(4 * crfD + 3)] ← c

if (frA(ps1) is an SNaN or frB(ps1) is an SNaN )
then VXSNAN ← 1
```

The floating-point operand in **fr**A(ps1) is compared to the floating-point operand in **fr**B(ps1). The result of the compare is placed into CR field **crf**D and the FPCC.

If one of the operands is a NaN, either quiet or signaling, then CR field **crf**D and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, then VXSNAN is set.

Other registers altered: (exception conditions are based on ps1 values)

- Condition Register (CR field specified by operand **crf**D):

  Affected: LT, GT, EQ, UN

- Floating-Point Status and Control Register:

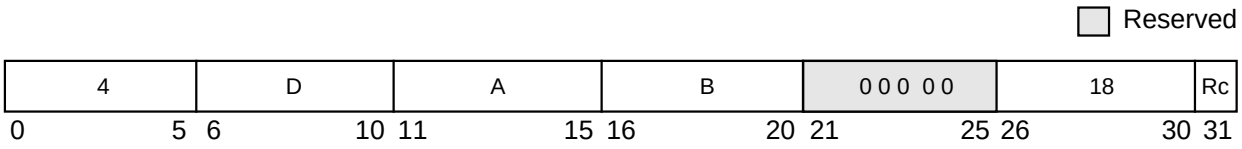
  Affected: FPCC(ps1 only), FX, VXSNAN

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		X

ps_div*x* ps_div*x*

Paired Single Divide (x'1000 0024')

ps_div **fr**D**,fr**A**,fr**B (Rc = 0)
ps_div. **fr**D**,fr**A**,fr**B (Rc = 1)



The following operations are performed:

```
If HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← frA(ps0) ÷ frB(ps0)
frD(ps1) ← frA(ps1) ÷ frB(ps1)
```

The floating-point operand in register **fr**A(ps0) is divided by the floating-point operand in register **fr**B(ps0). The remainder is not supplied as a result. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps0).

The floating-point operand in register **fr**A(ps1) is divided by the floating-point operand in register **fr**B(ps1). The remainder is not supplied as a result. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps1).

Floating-point division is based on exponent subtraction and division of the significands.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = 1 and zero divide exceptions when FPSCR[ZE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

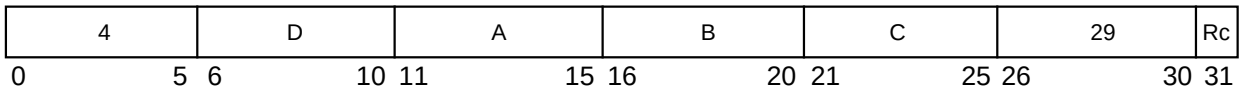
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control register (FPSCR):
  Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, ZX, XX, VXSNAN, VXIDI, VXZDZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_madd*x* ps_madd*x*

Paired Single Multiply-Add (x'1000 003A')

ps_madd **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
ps_madd. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operations are performed:

```
If HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← [frA(ps0) * frC(ps0)] + frB(ps0)
frD(ps1) ← [frA(ps1) * frC(ps1)] + frB(ps1)
```

The floating-point operand in register **fr**A(ps0) is multiplied by the floating-point operand in register **fr**C(ps0). The floating-point operand in register **fr**B(ps0) is added to this intermediate product. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps0).

The floating-point operand in register **fr**A(ps1) is multiplied by the floating-point operand in register **fr**C(ps1). The floating-point operand in register **fr**B(ps1) is added to this intermediate product. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps1).

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

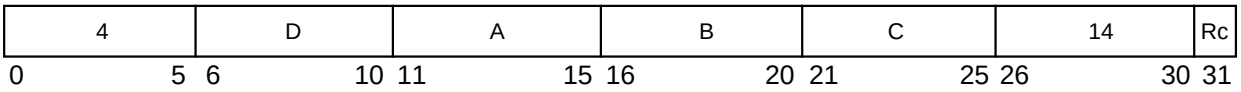
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control register(FPSCR):
  Affected: FPRF(ps0 only), FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_madds0*x* ps_madds0*x*

Paired Single Multiply-Add Scalar high(x'1000 001C')

ps_madds0 **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
ps_madds0. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← [frA(ps0) * frC(ps0)] + frB(ps0)
frD(ps1) ← [frA(ps1) * frC(ps0)] + frB(ps1)
```

The floating-point operand in register **fr**A(ps0) is multiplied by the floating-point operand in register **fr**C(ps0). The floating-point operand in register **fr**B(ps0) is added to this intermediate result, if the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and is placed into **fr**D(ps0).

The floating-point operand in register **fr**A(ps1) is multiplied by the floating-point operand in register **fr**C(ps0). The floating-point operand in register **fr**B(ps1) is added to this intermediate result, if the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and is placed into **fr**D(ps1).

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

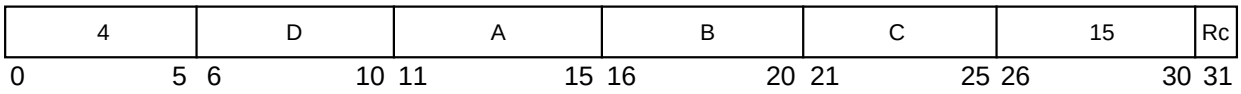
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control Register:
  Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_madds1*x* ps_madds1*x*

Paired Single Multiply-Add Scalar low(x'1000 001E')

ps_madds1 **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
ps_madds1. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← [frA(ps0) * frC(ps1)] + frB(ps0)
frD(ps1) ← [frA(ps1) * frC(ps1)] + frB(ps1)
```

The floating-point operand in register **fr**A(ps0) is multiplied by the floating-point operand in register **fr**C(ps1). The floating-point operand in register **fr**B(ps0) is added to this intermediate product. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps0) .

The floating-point operand in register **fr**A(ps1) is multiplied by the floating-point operand in register **fr**C(ps1). The floating-point operand in register **fr**B(ps1) is added to this intermediate product. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps1) .

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control Register:
  Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

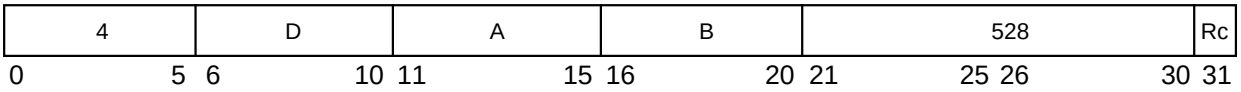
PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_merge00*x* ps_merge00*x*

Paired Single MERGE high (x'1000 0420')

ps_merge00 **fr**D**,fr**A**,fr**B (Rc = 0)

ps_merge00. **fr**D**,fr**A**,fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← frA(ps0)
frD(ps1) ← frB(ps0)
```

The floating-point operand in register **fr**A(ps0) is moved to register **fr**D(ps0) and floating-point operand in register **fr**B(ps0) is moved to register **fr**D(ps1).

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

- Condition Register (CR1 field):

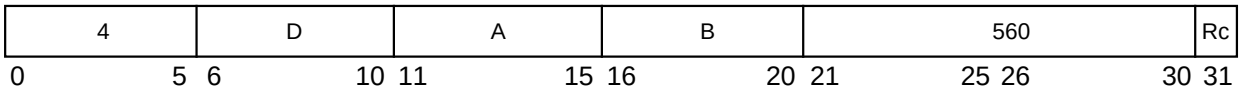
  Affected: FX, FEX, VX, OX (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		X

ps_merge01*x* ps_merge01*x*

Paired Single MERGE direct(x'1000 0460')

ps_merge01 **fr**D,**fr**A,**fr**B (Rc = 0)
ps_merge01. **fr**D,**fr**A,**fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← frA(ps0)
frD(ps1) ← frB(ps1)
```

The floating-point operand in register **fr**A(ps0) is moved to register **fr**D(ps0) and floating-point operand in register **fr**B(ps1) is moved to register **fr**D(ps1).

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

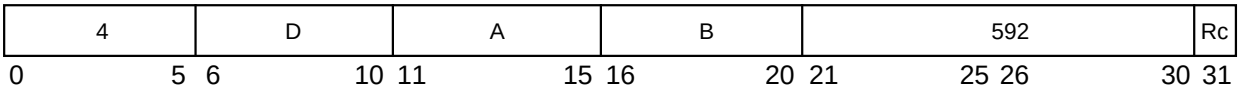
PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		X

ps_merge10*x* ps_merge10*x*

Paired Single MERGE swapped(x'1000 04A0')

ps_merge10 **fr**D**,fr**A**,fr**B (Rc = 0)

ps_merge10. **fr**D**,fr**A**,fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← frA(ps1)
frD(ps1) ← frB(ps0)
```

The floating-point operand in register **fr**A(ps1) is moved to register **fr**D(ps0) and floating-point operand in register **fr**B(ps0) is moved to register **fr**D(ps1).

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

- Condition Register (CR1 field):

  Affected: FX, FEX, VX, OX (if Rc = 1)

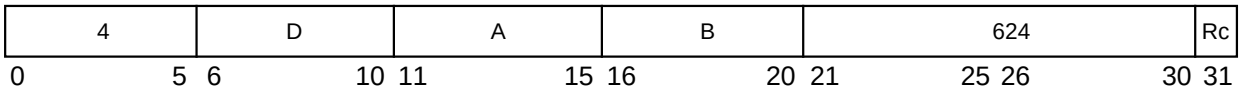
PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		X

ps_merge11*x* ps_merge11*x*

Paired Single MERGE low(x'1000 04E0')

ps_merge11 **fr**D,**fr**A,**fr**B (Rc = 0)

ps_merge11. **fr**D,**fr**A,**fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← frA(ps1)
frD(ps1) ← frB(ps1)
```

The floating-point operand in register **fr**A(ps1) is moved to register **fr**D(ps0) and floating-point operand in register **fr**B(ps1) is moved to register **fr**D(ps1).

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

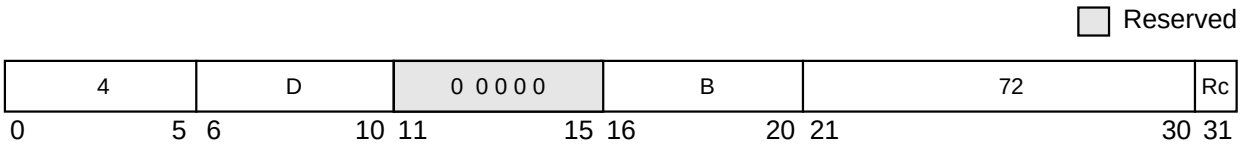
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		X

ps_mrx ps_mrx

Paired Single Move Register (x'1000 0090')

ps_mr **fr**D,**fr**B (Rc = 0)
ps_mr. **fr**D,**fr**B (Rc = 1)



The following operations are performed:

```
If HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← frB(ps0)
frD(ps1) ← frB(ps1)
```

The contents of register **fr**B(ps0) are placed into **fr**D(ps0).
The contents of register **fr**B(ps1) are placed into **fr**D(ps1).

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

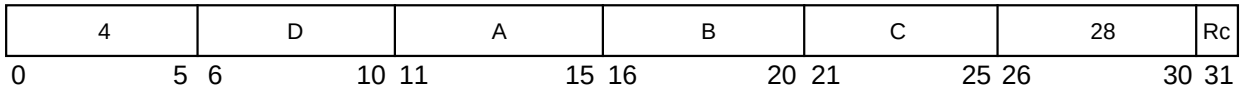
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		X

ps_msub*x* ps_msub*x*

Paired Single Multiply-Subtract (x'1000 0038')

ps_msub **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
ps_msub. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0 then invoke the illegal instruction error handler frD(ps0)
← [frA(ps0) * frC(ps0)] - frB(ps0)
frD(ps1) ← [frA(ps1) * frC(ps1)] - frB(ps1)
```

The floating-point operand in register **fr**A(ps0) is multiplied by the floating-point operand in register **fr**C(ps0). The floating-point operand in register **fr**B(ps0) is subtracted from this intermediate product. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps0).

The floating-point operand in register **fr**A(ps1) is multiplied by the floating-point operand in register **fr**C(ps1). The floating-point operand in register **fr**B(ps1) is subtracted from this intermediate product. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps1).

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

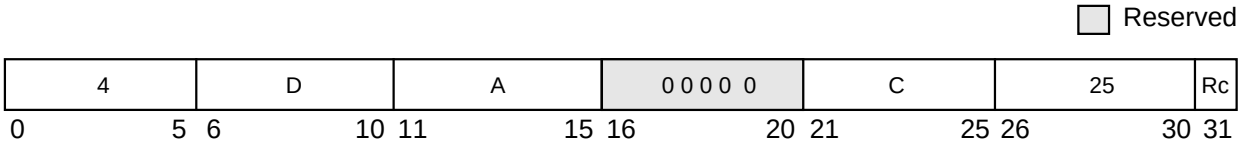
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control register (FPSCR):
  Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_mulx ps_mulx

Paired Single Multiply (x'1000 0032')

ps_mul **fr**D**,fr**A**,fr**C (Rc = 0)
ps_mul. **fr**D**,fr**A**,fr**C (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← frA(ps0) * frC(ps0)
frD(ps1) ← frA(ps1) * frC(ps1)
```

The floating-point operand in register **fr**A(ps0) is multiplied by the floating-point operand in register **fr**C(ps0). If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps0).

The floating-point operand in register **fr**A(ps1) is multiplied by the floating-point operand in register **fr**C(ps1). If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps1).

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control register (FPSCR):
  Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNAN, VXIMZ

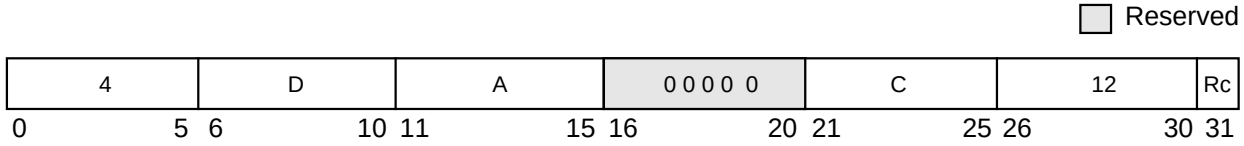
PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_muls0*x* ps_muls0*x*

Paired Single Multiply Scalar high(x'1000 0018')

ps_muls0 **fr**D,**fr**A,**fr**C (Rc = 0)

ps_muls0. **fr**D,**fr**A,**fr**C (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← frA(ps0) * frC(ps0)
frD(ps1) ← frA(ps1) * frC(ps0)
```

The floating-point operand in register **fr**A(ps0) is multiplied by the floating-point operand in register **fr**C(ps0). If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps0).

The floating-point operand in register **fr**A(ps1) is multiplied by the floating-point operand in register **fr**C(ps0). If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps1).

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control Register (FPSCR):
  Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

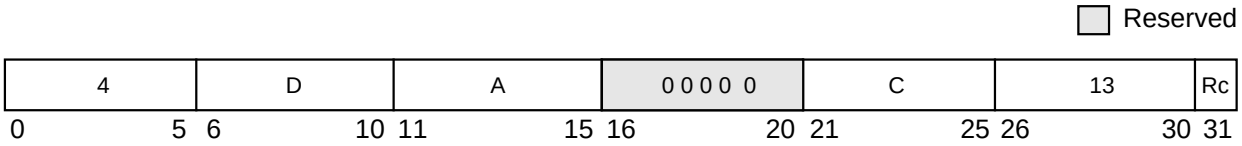
PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_muls1*x* ps_muls1*x*

Paired Single Multiply Scalar low(x'1000 001A')

ps_muls1 **fr**D**,fr**A**,fr**C (Rc = 0)

ps_muls1. **fr**D**,fr**A**,fr**C (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← frA(ps0) * frC(ps1)
frD(ps1) ← frA(ps1) * frC(ps1)
```

The floating-point operand in register **fr**A(ps0) is multiplied by the floating-point operand in register **fr**C(ps1). If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and are placed into **fr**D(ps0).

The floating-point operand in register **fr**A(ps1) is multiplied by the floating-point operand in register **fr**C(ps1). If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and are placed into **fr**D(ps1).

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

- Condition Register (CR1 field):

  Affected: FX, FEX, VX, OX (if Rc = 1)

- Floating-Point Status and Control Register (FPSCR):

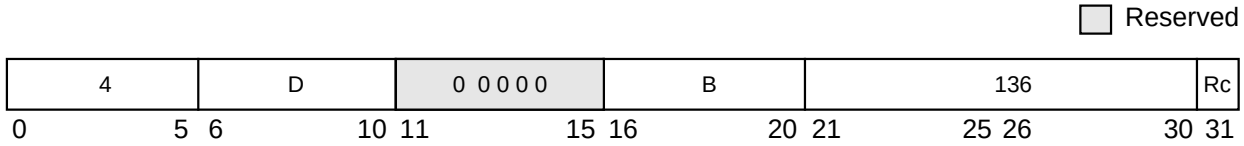
  Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_nabs*x* ps_nabs*x*

Paired Single Negative Absolute Value (x'1000 0110')

ps_nabs **fr**D**,fr**B (Rc = 0)
ps_nabs. **fr**D**,fr**B (Rc = 1)



The following operations are performed:

```
If HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← b'1' || frB(ps0)[1-31]
frD(ps1) ← b'1' || frB(ps1)[1-31]
```

The contents of register **fr**B(ps0) with bit 0 set are placed into **fr**D(ps0).
The contents of register **fr**B(ps1) with bit 0 set are placed into **fr**D(ps1).

NOTE: The **ps_nabs** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN may be altered by **ps_nabs**. This instruction does not alter the FPSCR.

Other registers altered:

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

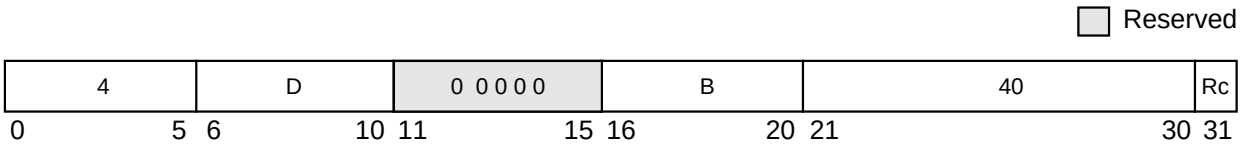
PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		X

ps_negx

ps_negx

Paired Single Negate (x'1000 0050')

ps_neg **fr**D,**fr**B (Rc = 0)
ps_neg. **fr**D,**fr**B (Rc = 1)



The following operations are performed:

```
If HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← ¬(frB(ps0)[0] || frB(ps0)[1-31] )
frD(ps1) ← ¬(frB(ps1)[0] || frB(ps1)[1-31] )
```

The contents of register **fr**B(ps0) with bit 0 inverted are placed into **fr**D(ps0).
The contents of register **fr**B(ps1) with bit 0 inverted are placed into **fr**D(ps1).

Note that the **ps_neg** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN may be altered by **ps_neg**. This instruction does not alter the FPSCR.

Other registers altered:

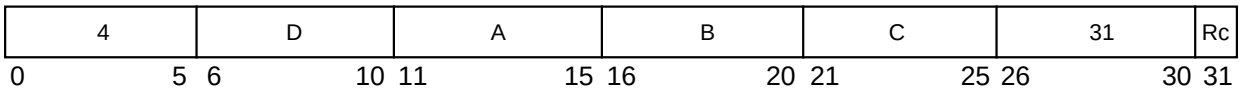
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		X

ps_nmadd*x* ps_nmadd*x*

Paired Single Negative Multiply-Add (x'1000 003E')

ps_nmadd **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
ps_nmadd. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← -[frA(ps0) * frC(ps0) + frB(ps0) ]
frD(ps1) ← -[frA(ps1) * frC(ps1) + frB(ps1) ]
```

The floating-point operand in register **fr**A(ps0) is multiplied by the floating-point operand in register **fr**C(ps0).

The floating-point operand in register **fr**B(ps0) is added to this intermediate product and the result is negated.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps0).

The floating-point operand in register **fr**A(ps1) is multiplied by the floating-point operand in register **fr**C(ps1).

The floating-point operand in register **fr**B(ps1) is added to this intermediate product and the result is negated.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps1).

This instruction produces the same result as would be obtained by using the Paired Single Multiply-Add (**ps_madd***x*) instruction and then negating the result, with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control register (FPSCR):
  Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

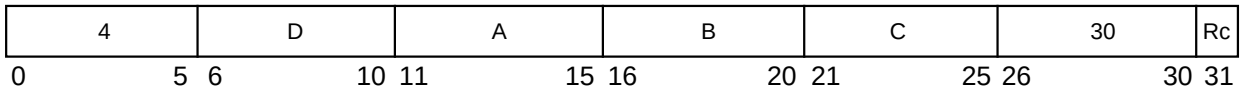
PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_nmsub*x* ps_nmsub*x*

Paired Single Negative Multiply-Subtract (x'1000 003C')

ps_nmsub **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)

ps_nmsub. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← -[frA(ps0) * frC(ps0) - frB(ps0) ]
frD(ps1) ← -[frA(ps1) * frC(ps1) - frB(ps1) ]
```

The floating-point operand in register **fr**A(ps0) is multiplied by the floating-point operand in register **fr**C(ps0). The floating-point operand in register **fr**B(ps0) is subtracted from this intermediate product. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **fr**D(ps0).

The floating-point operand in register **fr**A(ps1) is multiplied by the floating-point operand in register **fr**C(ps1). The floating-point operand in register **fr**B(ps1) is subtracted from this intermediate product. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **fr**D(ps1).

This instruction produces the same result obtained by negating the result of a Floating Multiply-Subtract (**ps_msub***x*) instruction with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

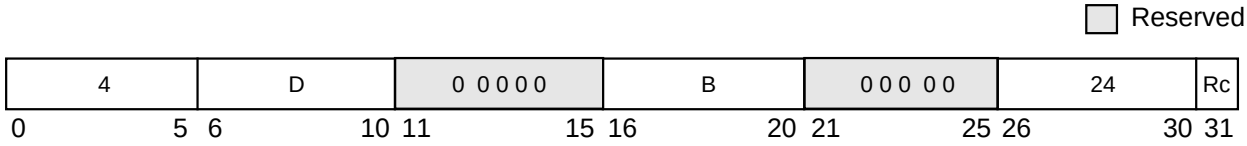
- Condition Register (CR1 field)
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control register (FPSCR):
  Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_resx ps_resx

Paired Single Reciprocal Estimate (x'1000 0030')

ps_res **fr**D,**fr**B (Rc = 0)
ps_res. **fr**D,**fr**B (Rc = 1)



A single-precision estimate of the reciprocal of the floating-point operand in register **fr**B(ps0) is placed into register **fr**D(ps0) and a single-precision estimate of the reciprocal of the floating-point operand in register **fr**B(ps1) is placed into register **fr**D(ps1). These estimates placed into register **fr**D(ps0) and **fr**D(ps1) are correct to a precision of one part in 4096 of the reciprocal of **fr**B(ps0) and **fr**B(ps1), respectively. That is, for each calculation:

$$\mathrm{ABS}\left(\frac{\text{estimate}-\left(\frac{1}{x}\right)}{\left(\frac{1}{x}\right)}\right) \leq \frac{1}{4096}$$

where x is the **fr**B(ps0) or **fr**B(ps1) value in the source registers.

Operation with various special values of the operand is summarized below:

Operand	ResultException	
–∞	–0	None
–0	–∞*ZX	
+0	+∞*ZX	
+∞	+0None	
SNaN	QNaN**VXSNAN	
QNaN	QNaNNone	

Notes: * No result if FPSCR[ZE] = 1
** No result if FPSCR[VE] = 1

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = 1 and zero divide exceptions when FPSCR[ZE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

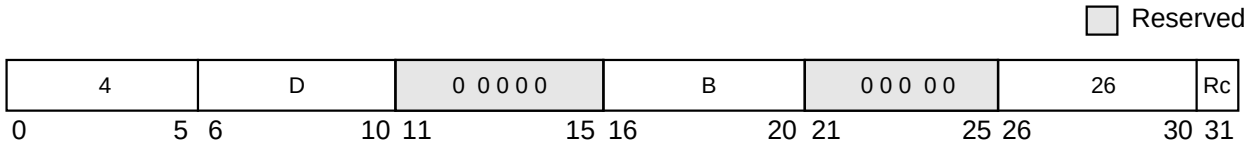
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control register (FPSCR):
  Affected: FPRF (ps0 only), FR (undefined), FI (undefined), FX, OX, UX, ZX, VXSNAN

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_rsqrtex ps_rsqrtex

Paired Single Reciprocal Square Root Estimate (x'1000 0034')

ps_rsqrte	**fr**D,**fr**B	(Rc = 0)
ps_rsqrte.	**fr**D,**fr**B	(Rc = 1)



A single-precision estimate of the reciprocal of the square root of the floating-point operand in register **fr**B(ps0) is placed into register **fr**D(ps0). A single-precision estimate of the reciprocal of the square root of the floating-point operand in register **fr**B(ps1) is placed into register **fr**D(ps1).These estimates placed into register **fr**D(ps0) and **fr**D(ps1) are correct to a precision of one part in 4096 of the reciprocal of the square root of **fr**B(ps0) and **fr**B(ps1), respectively. That is, for each calculation:

$$\mathrm{ABS}\left(\frac{\text{estimate}-\left(\frac{1}{\sqrt{x}}\right)}{\left(\frac{1}{\sqrt{x}}\right)}\right) \leq \frac{1}{4096}$$

where x is the **fr**B(ps0) or **fr**B(ps1) value in the source registers.

Operations with various special values of the operand is summarized below:

Operand	ResultException
–∞	QNaN**VXSQRT
<0	QNaN**VXSQRT
–0	–∞*ZX
+0	+∞*
∞*	ZX
+∞	+0None
SNaN	QNaN**VXSNAN
QNaN	QNaNNone

Notes: * No result if FPSCR[ZE] = 1

** No result if FPSCR[VE] = 1

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = 1 and zero divide exceptions when FPSCR[ZE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

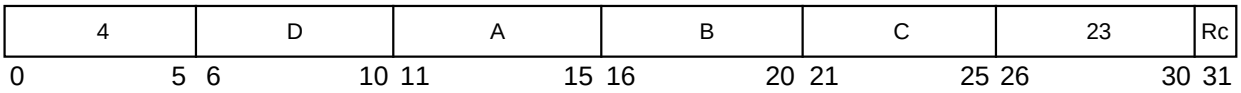
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control register (FPSCR):
  Affected: FPRF (ps0 only), FR (undefined), FI (undefined), FX, ZX, VXSNAN, VXSQRT

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_sel*x* ps_sel*x*

Paired Single Select (x'1000 002E')

ps_sel **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
ps_sel. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operations are performed:

```
If HID2[PSE] = 0 then invoke the illegal instruction error handler
if (frA(ps0) ≥ 0.0 )
  then frD(ps0) ← frC(ps0)
  else frD(ps0) ← frB(ps0)
if (frA(ps1) ≥ 0.0 )
  then frD(ps1) ← frC(ps1)
  else frD(ps1) ← frB(ps1)
```

The floating-point operand in register **fr**A(ps0) is compared to the value zero. If the operand is greater than or equal to zero, register **fr**D(ps0) is set to the contents of register **fr**C(ps0). If the operand is less than zero or is a NaN, register **fr**D(ps0) is set to the contents of register **fr**B(ps0).

The floating-point operand in register **fr**A(ps1) is compared to the value zero. If the operand is greater than or equal to zero, register **fr**D(ps1) is set to the contents of register **fr**C(ps1). If the operand is less than zero or is a NaN, register **fr**D(ps1) is set to the contents of register **fr**B(ps1).

These comparisons ignore the sign of zero (that is, regard +0 as equal to –0).

Care must be taken in using **ps_sel** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

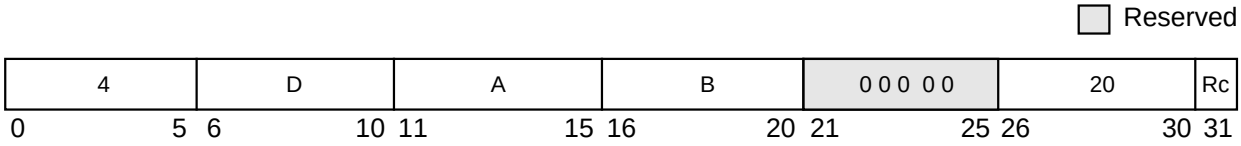
PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_subx

ps_subx

Paired Single Subtract (x'1000 0028')

ps_sub **fr**D**,fr**A**,fr**B (Rc = 0)
ps_sub. **fr**D**,fr**A**,fr**B (Rc = 1)



The following operations are performed:

```
If HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← frA(ps0) - frB(ps0)
frD(ps1) ← frA(ps1) - frB(ps1)
```

The floating-point operand in register **fr**B(ps0) is subtracted from the floating-point operand in register **fr**A(ps0). If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps0).

The floating-point operand in register **fr**B(ps1) is subtracted from the floating-point operand in register **fr**A(ps1). If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps1).

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

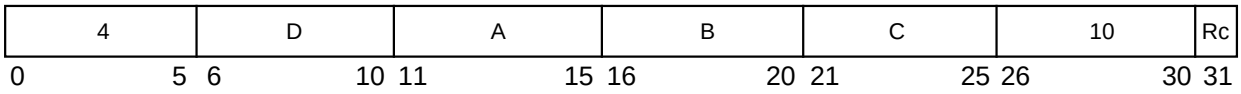
- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control register (FPSCR):
  Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNAN, VXISI

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_sum0*x* ps_sum0*x*

Paired Single vector SUM high (x'1000 0014')

ps_sum0 **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
ps_sum0. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD(ps0) ← frA(ps0) + frB(ps1)
frD(ps1) ← frC(ps1)
```

The floating-point operand in register **fr**A(ps0) is added to the floating-point operand from register **fr**B(ps1). If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps0).

The floating-point operand in register **fr**C(ps1) is placed into **fr**D(ps1).

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

- Condition Register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-Point Status and Control Register:
  Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNAN, VXISI

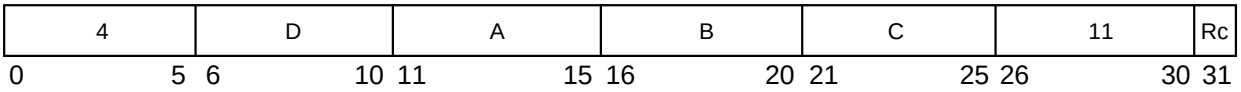
PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

ps_sum1*x* ps_sum1*x*

Paired Single vector SUM low(x'1000 0016')

ps_sum1 **fr**D**,fr**A**,fr**C**,fr**B (Rc = 0)

ps_sum1. **fr**D**,fr**A**,fr**C**,fr**B (Rc = 1)



The following operations are performed:

```
if HID2[PSE] = 0 then Goto  illegal instruction error handler
frD(ps0) ← frC(ps0)
frD(ps1) ← frA(ps0) + frB(ps1)
```

The floating-point operand in register **fr**C(ps0) is placed into **fr**D(ps0).

The floating-point operand in register **fr**A(ps0) is added to the floating-point operand from register **fr**B(ps1). If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D(ps1).

FPSCR[FPRF] is set to the class and sign of the ps1 result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered: (exception conditions are based on either ps0 or ps1 values)

- Condition Register (CR1 field):

  Affected: FX, FEX, VX, OX (if Rc = 1)

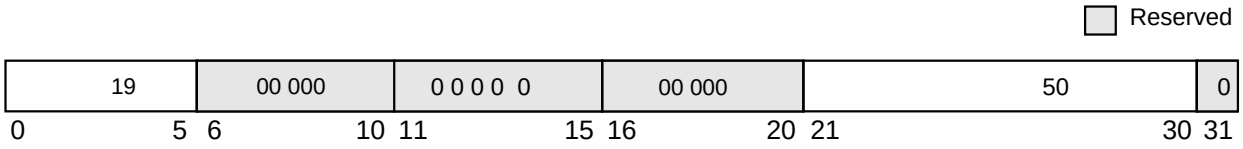
- Floating-Point Status and Control Register:

  Affected: FPRF (ps1 only), FR, FI, FX, OX, UX, XX, VXSNAN, VXISI

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA		Yes		A

rfi rfi

Return from Interrupt (x'4C00 0064')




```
MSR[0,5-9,16–23, 25–27, 30–31] ← SRR1[0,5-9,16–23, 25–27, 30–31]
MSR[13] ← b'0'
NIA ←iea SRR0[0–29] || 0b00
```

Bits SRR1[0,5-9,16–23, 25–27, 30–31] are placed into the corresponding bits of the MSR. MSR[13] is set to 0. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address SRR0[0–29] || 0b00. If the new MSR value enables one or more pending exceptions, the exception associated with the highest priority pending exception is generated; in this case the value placed into SRR0 by the exception processing mechanism is the address of the instruction that would have been executed next had the exception not occurred. Note that an implementation may define additional MSR bits, and in this case, may also cause them to be saved to SRR1 from MSR on an exception and restored to MSR from SRR1 on an **rfi**.

This is a supervisor-level, context synchronizing instruction.

Other registers altered:

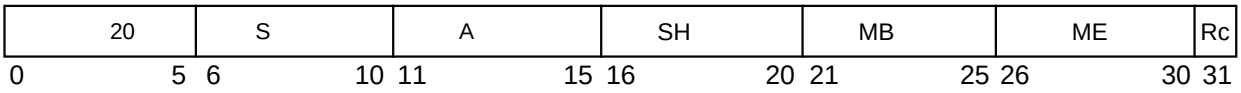
- MSR

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
OEA	YES			XL

rlwimi*x* rlwimi*x*

Rotate Left Word Immediate then Mask Insert (x'5000 0000')

rlwimi **r**A,**r**S,SH,MB,ME (Rc = 0)
rlwimi. **r**A,**r**S,SH,MB,ME (Rc = 1)




```
n ← SH
r ← ROTL(rS, n)
m ← MASK(MB, ME)
rA ← (r & m) | (rA & ¬ m)
```

The contents of **r**S are rotated left the number of bits specified by operand SH. A mask is generated having 1 bits from bit MBthrough bit ME and 0 bits elsewhere. The rotated data is inserted into **r**A under control of the generated mask.

NOTE: **rlwimi** can be used to copy a bit field of any length from register **r**S into the contents of **r**A. This field can start from any bit position in **r**S and be placed into any position in **r**A. The length of the field can range from 0 to 32 bits. The remaining bits in register **r**A remain unchanged. :

- To copy byte_0 (bits 0-7) from rS into byte_3 (bits 24-31) of rA , set SH = 8 , set MB = 24, and set ME = 31.
- In general, to copy an *n*-bit field that starts in bit position b in register rS into register rA starting a bit position c: set SH = 32 - c + b Mod(32), set MB = *c*, and set ME = (*c* + *n*) – 1 Mod(32).

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

Simplified mnemonics:

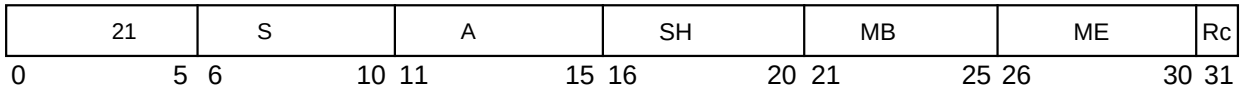
inslwi rA,rS,n,b	equivalent to	rlwimi rA,rS,32 – b,b,b + n – 1
insrwi rA,rS,n,b (n > 0)	equivalent to	rlwimi rA,rS,32 – (b + n),b,(b + n) – 1

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				M

rlwinm*x* rlwinm*x*

Rotate Left Word Immediate then AND with Mask (x'5400 0000')

rlwinm **r**A,**r**S,SH,MB,ME (Rc = 0)
rlwinm. **r**A,**r**S,SH,MB,ME (Rc = 1)




```
n ← SH
r ← ROTL(rS, n)
m ← MASK(MB, ME)
rA ← r & m
```

The contents of **r**S are rotated left the number of bits specified by operand SH. A mask is generated having 1 bits from bit MBthrough bit MEand 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

NOTE: **rlwinm** can be used to extract, rotate, shift, and clear bit fields using the methods shown below:

- To extract an *n*-bit field, that starts at bit position *b* in **r**S, right-justified into **r**A (clearing the remaining 32 – *n* bits of **r**A), set SH = *b* + *n*, set MB = 32 – *n*, and set ME = 31.
- To extract an *n*-bit field, that starts at bit position *b* in **r**S, left-justified into **r**A (clearing the remaining 32 – *n* bits of **r**A), set SH = *b*, set MB = 0, and set ME = *n* – 1.
- To rotate the contents of a register left (or right) by *n* bits, set SH = *n* (32 – *n*), set MB = 0, and set ME = 31.
- To shift the contents of a register right by *n* bits, by setting SH = 32 – *n*, MB = *n*, and ME = 31.

  It can also be used to clear the high-order *b* bits of a register and then shift the result left by *n* bits by setting SH = *n*, by setting MB = *b* – *n*, and by setting ME = 31 – *n*.
- To clear the low-order *n* bits of a register, by setting SH = 0, MB = 0, and ME = 31 – *n*.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

Simplified mnemonics:

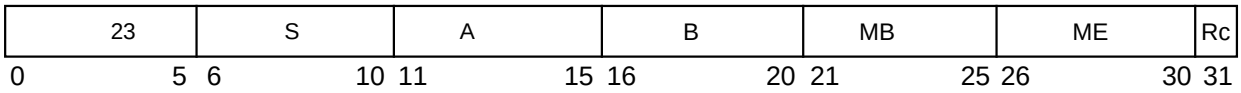
extlwi rA**,r**S**,***n***,***b** (*n* > 0)	equivalent to	**rlwinm r**A**,r**S,b**,0,***n* – 1
extrwi rA**,r**S**,***n***,***b** (*n* > 0)	equivalent to	**rlwinm r**A**,r**S,b + ***n***,**32 – ***n***,31**
rotlwi rA**,r**S**,***n*	equivalent to	**rlwinm r**A**,r**S**,***n***,0,31**
rotrwi rA**,r**S**,***n*	equivalent to	**rlwinm r**A**,r**S**,32 –** ***n***,0,31**
slwi rA**,r**S**,***n* (*n* < 32)	equivalent to	**rlwinm r**A**,r**S**,***n***,0,**31**–*n*
srwi rA**,r**S**,***n* (*n* < 32)	equivalent to	**rlwinm r**A**,r**S,32 – ***n***,***n***,31**
clrlwi rA**,r**S**,***n* (*n* < 32)	equivalent to	**rlwinm r**A**,r**S**,0,***n***,31**
clrrwi rA**,r**S**,***n* (*n* < 32)	equivalent to	**rlwinm r**A**,r**S**,0,0,**31 – *n*
clrlslwi rA**,r**S**,***b***,***n* (*n* *b* < 32)	equivalent to	**rlwinm r**A**,r**S**,***n***,***b* – ***n***,**31 – *n*

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				M

rlwnm*x* rlwnm*x*

Rotate Left Word then AND with Mask (x'5C00 0000')

rlwnm **r**A,**r**S,**r**B,MB,ME (Rc = 0)
rlwnm. **r**A,**r**S,**r**B,MB,ME (Rc = 1)




```
n ← rB[27-31]
r ← ROTL(rS, n)
m ← MASK(MB, ME)
rA ← r & m
```

The contents of **r**S are rotated left the number of bits specified by the low-order five bits of **r**B. A mask is generated having 1 bits from bit MB through bit ME and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

NOTE: **rlwnm** can be used to extract and to rotate bit fields using one of these methods:

- To extract an *n*-bit field, that starts at variable bit position *b* in **r**S, right-justified into **r**A (clearing the remaining 32 – *n* bits of **r**A), set the low-order five bits of **r**B to *b* + *n*, set MB = 32 – *n*, and set ME = 31.
- To extract an *n*-bit field, that starts at variable bit position *b* in **r**S, left-justified into **r**A (clearing the remaining 32 – *n* bits of **r**A), set the low-order five bits of **r**B to *b*, set MB = 0, and set ME = *n* – 1.
- To rotate the contents of a register left (or right) by *n* bits, set the low-order five bits of **r**B to *n* (32 – *n*), set MB = 0, and set ME = 31.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

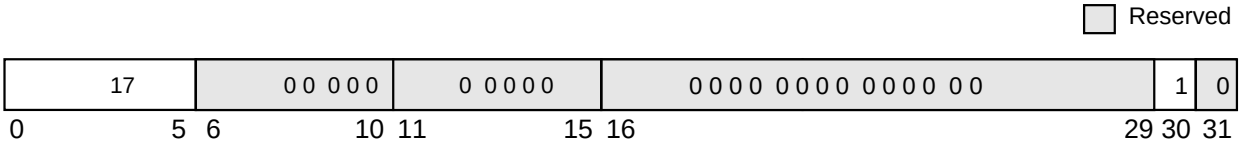
Simplified mnemonics:

rotlwrA,**r**S,**r**B equivalent to **rlwnm r**A,**r**S,**r**B**,0,31**

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				M

SC SC

System Call (x'4400 0002')



In the PowerPC UISA, the **sc** instruction calls the operating system to perform a service. When control is returned to the program that executed the system call, the content of the registers depends on the register conventions used by the program providing the system service.

This instruction is context synchronizing, as described in Section 4.1.5.1, "Context Synchronizing Instructions," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Other registers altered:

- Dependent on the system service

In PowerPC OEA, the **sc** instruction does the following:

```
SRR0 ←iea CIA + 4
SRR1[1-41-4, 10-151] ← 0
SRR1[0,5-9, 16-23, 25-27, 30-31] ← MSR[0,5-9, 16-23, 25-27, 30-31]
MSR ← new_value (see below)
NIA ←iea base_ea + 0xC00 (see below)
```

The EA of the instruction following the **sc** instruction is placed into SRR0. Bits 0, 5-9,16-23, 25-27, and 30-31 of the MSR are placed into the corresponding bits of SRR1, and bits 1-4 and 10-15 of SRR1 are set to undefined values.

NOTE: An implementation may define additional MSR bits, and in this case, may also cause them to be saved to SRR1 from MSR on an exception and restored to MSR from SRR1 on an **rfi**; then a system call exception is generated. The exception causes the MSR to be altered as described in Section 6.4, "Exception Definitions" in *The Programming Environments Manual.*

The exception causes the next instruction to be fetched from offset 0xC00 from the physical base address determined by the new setting of MSR[IP].

Other registers altered:

- SRR0
- SRR1
- MSR

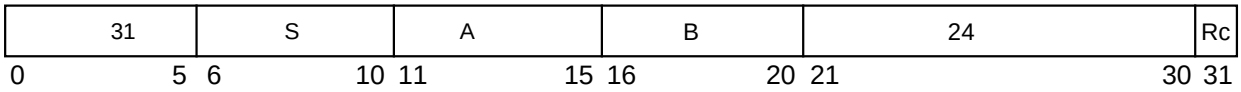
PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA/OEA				SC

slw*x* slw*x*

Shift Left Word (x'7C00 0030')

slw **r**A,**r**S,**r**B (Rc = 0)

slw. **r**A,**r**S,**r**B (Rc = 1)




```
n← rB[27-31]
r ← ROTL(rS , n)
if rB[26] = 0
then m ← MASK(0 , 31 – n)
else m ← (32)0
rA ← r & m
```

The contents of **r**S are shifted left the number of bits specified by the low-order five bits of **r**B. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into **r**A. However, shift amounts from 32 to 63 give a zero result.

Other registers altered:

- Condition Register (CR0 field):

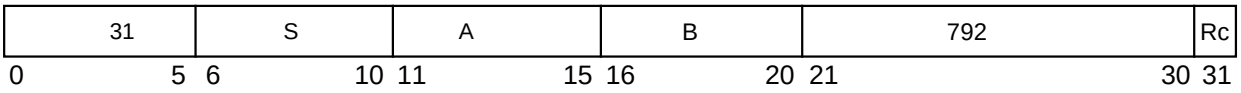
  Affected: LT, GT, EQ, SO (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

srawx srawx

Shift Right Algebraic Word (x'7C00 0630')

sraw	**r**A,**r**S,**r**B	(Rc = 0)
sraw.	**r**A,**r**S,**r**B	(Rc = 1)




```
n← rB[27-31]
 r ← ROTL(rS, 32– n)
 if rB[26] = 0
 then m ← MASK(n, 31)
 else m ← (32)0
 S ← rS(0)
 rA ← r & m | (32)S & ¬ m
 XER[CA] ← S & (r & ¬ m[0-31] ≠ 0
```

The contents of **r**S are shifted right the number of bits specified by the low-order five bits of **r**B (shift amounts between 0-31). Bits shifted out of position 31 are lost. Bit 0 of **r**S is replicated to fill the vacated positions on the left. The 32-bit result is placed into **r**A. XER[CA] is set if **r**S contains a negative number and any 1 bits are shifted out of position 31; otherwise XER[CA] is cleared. A shift amount of zero causes **r**A to receive the 32 bits of **r**S, and XER[CA] to be cleared. However, shift amounts from 32 to 63 give a result of 32 sign bits, and cause XER[CA] to receive the sign bit of **r**S.

NOTE: The **sraw** instruction, followed by **addze**, can be used to divide quickly by 2^n.

Other registers altered:

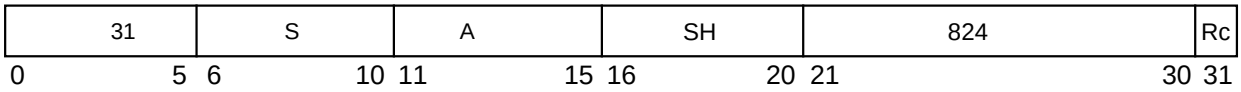
- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
- XER:
  Affected: CA

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

srawi*x* srawi*x*

Shift Right Algebraic Word Immediate (x'7C00 0670')

srawi **r**A,**r**S,SH (Rc = 0)
srawi. **r**A,**r**S,SH (Rc = 1)




```
n ← SH
r ← ROTL(rS, 32 – n)
m← MASK(n, 31)
S ← rS(0)
rA ← r & m | (32)S & ¬ m
XER[CA] ← S(0) & ((r & ¬ m) ≠ 0)
```

The contents of **r**S are shifted right SH bits. Bits shifted out of position 31 are lost. Bit 0 of **r**S is replicated to fill the vacated positions on the left. The result is placed into **r**A. XER[CA] is set if the 32 bits of **r**S contain a negative number and any 1 bits are shifted out of position 31; otherwise XER[CA] is cleared. A shift amount of zero causes **r**A to receive the value of **r**S, and XER[CA] to be cleared.

NOTE: The **srawi** instruction followed by **addze** instruction can be used to divide quickly by 2^n.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
- XER:
  Affected: CA

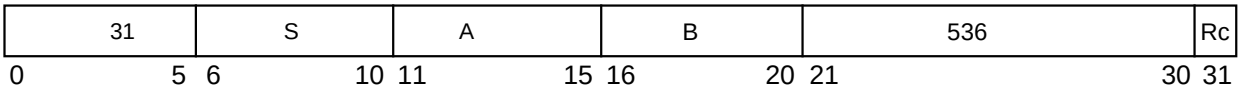
PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

srw*x* srw*x*

Shift Right Word (x'7C00 0430')

srw **r**A,**r**S,**r**B (Rc = 0)

srw. **r**A,**r**S,**r**B (Rc = 1)




```
n ← rB[27-31]
r ← ROTL(rS, 32–n)
if rB[26] = 0
then m ← MASK(n , 31)
else m ← (32)0
rA ← r & m
```

The contents of **r**S are shifted right the number of bits specified by the low-order five bits of **r**B (shift amounts between 0-31). Bits shifted out of position 31 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into **r**A. However, shift amounts from 32 to 63 give a zero result.

Other registers altered:

- Condition Register (CR0 field):

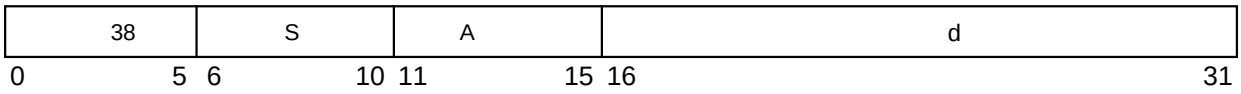
  Affected: LT, GT, EQ, SO (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

stb stb

Store Byte (x’9800 0000’)

stb **r**S,d**(r**A**)**




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 1) ← rS[24-31]
```

EA is the sum (**r**A|0) + d. The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by EA.

Other registers altered:

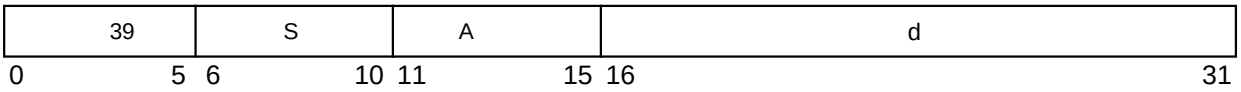
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

stbu stbu

Store Byte with Update (x'9C00 0000')

stbu **r**S,d(**r**A)




```
EA ← (rA) + EXTS(d)
MEM(EA, 1) ← rS[24-31]
rA ← EA
```

EA is the sum (**r**A) + d. The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

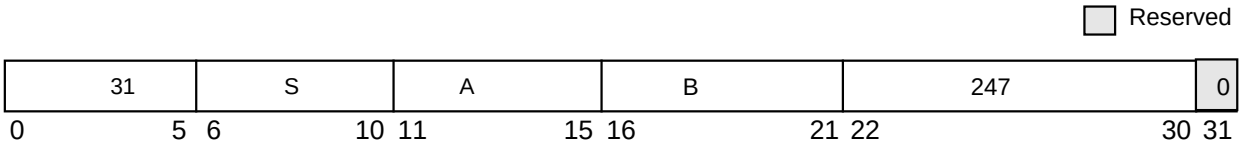
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

stbux stbux

Store Byte with Update Indexed (x'7C00 01EE')

stbux **r**S,**r**A,**r**B




```
EA ← (rA) + (rB)
MEM(EA, 1) ← rS[24-31]
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

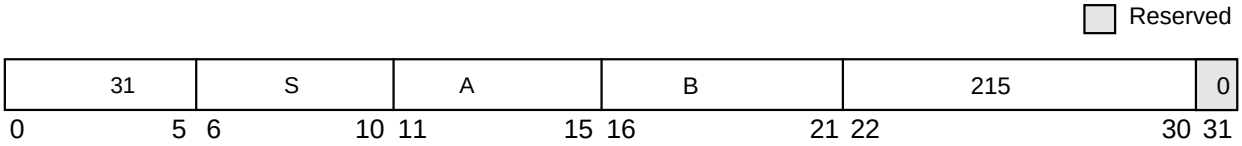
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

stbx stbx

Store Byte Indexed (x'7C00 01AE')

stbx **r**S,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA, 1) ← rS[24-31]
```

EA is the sum (**r**A|0) + (**r**B). The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by EA.

Other registers altered:

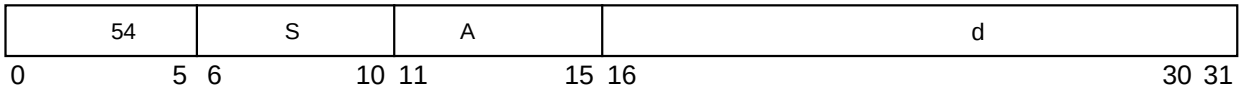
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

stfd stfd

Store Floating-Point Double (x'D800 0000')

stfd **fr**S,d**(r**A**)**




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 8) ← (frS)
```

EA is the sum (**r**A|0) + d.

The contents of register **fr**S are stored into the double word in memory addressed by EA.

Other registers altered:

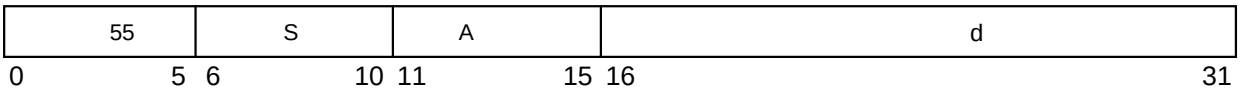
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

stfdu stfdu

Store Floating-Point Double with Update (x'DC00 0000')

stfdu **fr**S,d**(r**A**)**




```
EA ← (rA) + EXTS(d)
MEM(EA, 8) ← (frS)
rA ← EA
```

EA is the sum (**r**A) + d.

The contents of register **fr**S are stored into the double word in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

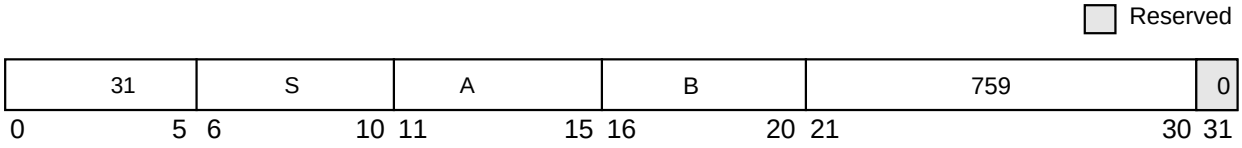
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

stfdux stfdux

Store Floating-Point Double with Update Indexed (x'7C00 05EE')

stfdux **fr**S,**r**A,**r**B



EA ← (**r**A) + (**r**B)
MEM(EA, 8) ← (**fr**S)
rA ← EA

EA is the sum (**r**A) + (**r**B).

The contents of register **fr**S are stored into the double word in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

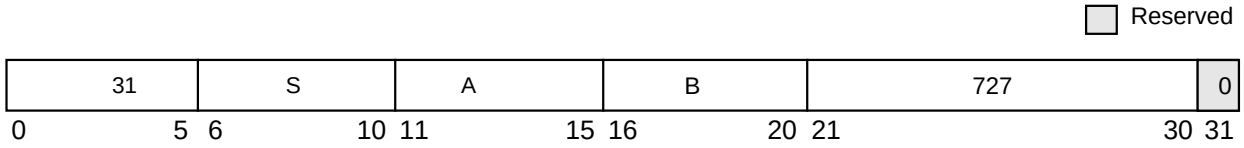
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

stfdx stfdx

Store Floating-Point Double Indexed (x'7C00 05AE')

stfdx **fr**S,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA, 8) ← (frS)
```

EA is the sum (**r**A|0) + **r**B.

The contents of register **fr**S are stored into the double word in memory addressed by EA.

Other registers altered:

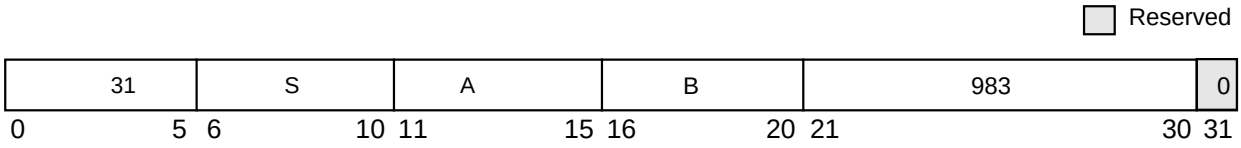
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

stfiwx stfiwx

Store Floating-Point as Integer Word Indexed (x'7C00 07AE')

stfiwx **fr**S,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← frS[32–63]
```

EA is the sum (**r**A|0) + (**r**B).

The contents of the low-order 32 bits of register **fr**S are stored, without conversion, into the word in memory addressed by EA.

This instruction when preceded by the floating-point convert to integer word (fctiwx) or floating-point convert to integer word with round toward zero (**fctiwzx**) will store the 32-bit integer value of a double-precision floating-point number. (see **fctiwx** and **fctiwzx** instructions)

Do NOT attempt to use this instruction to store the ps1 value for paired-single floating-point operands, the stored value is undefined.

If the content of register **fr**S is a double-precision floating point number, the low-order 32 bits of the 52 bit mantissa are stored. (without the exponent, this could be a meaningless value)

If the contents of register **fr**S were produced, either directly or indirectly, by an **lfs** instruction, a single-precision arithmetic instruction, or **frsp**, then the value stored is the low-order 32 bits of the 52 bit mantissa of the double-precision number. (all single-precision floating-point numbers are maintained in double precision format in the floating-point register file)

When HID2[PSE] = 1, the input operand in **fr**S must be the result of an **fctiw** or **fctiwz** instruction. Otherweise, the result is undefined.

Other registers altered:

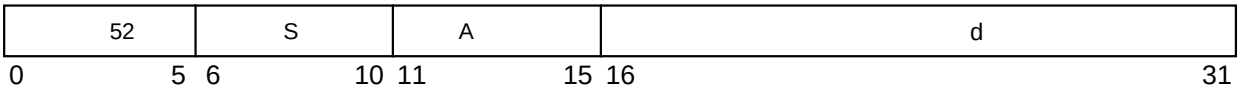
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA			YES	X

stfs stfs

Store Floating-Point Single (x'D000 0000')

stfs **fr**S,d**(r**A**)**




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 4) ← SINGLE(frS)
```

EA is the sum (**r**A|0) + d.

The contents of register **fr**S are converted to single-precision and stored into the word in memory addressed by EA. For a discussion on floating-point store conversions, see Section D.7, "Floating-Point Store Instructions."

Other registers altered:

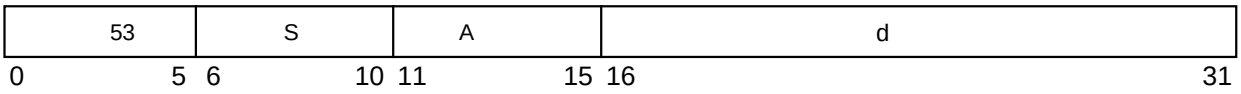
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

stfsu stfsu

Store Floating-Point Single with Update (x'D400 0000')

stfsu **fr**S,d**(r**A**)**




```
EA ← (rA) + EXTS(d)
MEM(EA, 4) ← SINGLE(frS)
rA ← EA
```

EA is the sum (**r**A) + d.

The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by EA. For a discussion on floating-point store conversions, see Section D.7, "Floating-Point Store Instructions," in *The Programming Environments Manual.*

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

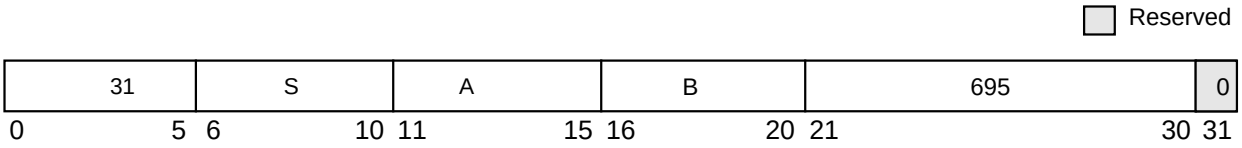
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

stfsux stfsux

Store Floating-Point Single with Update Indexed (x'7C00 056E')

stfsux **fr**S,**r**A,**r**B




```
EA ← (rA) + (rB)
MEM(EA, 4) ← SINGLE(frS)
rA ← EA
```

EA is the sum (**r**A) + (**r**B).

The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by EA. For a discussion on floating-point store conversions, see Section D.7, "Floating-Point Store Instructions," in *The Programming Environments Manual.*

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

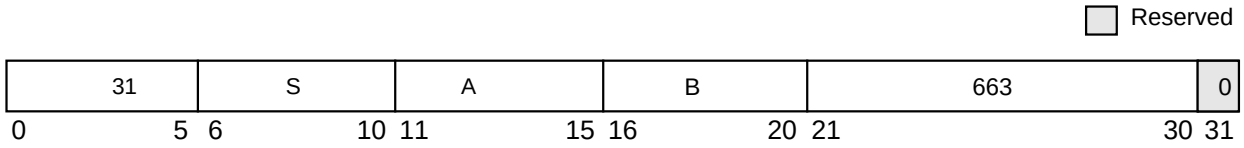
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

stfsx stfsx

Store Floating-Point Single Indexed (x'7C00 052E')

stfsx **fr**S,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← SINGLE(frS)
```

EA is the sum (**r**A|0) + (**r**B).

The contents of register **fr**S are converted to single-precision and stored into the word in memory addressed by EA. For a discussion on floating-point store conversions, see Section D.7, "Floating-Point Store Instructions," in *The Programming Environments Manual.*

Other registers altered:

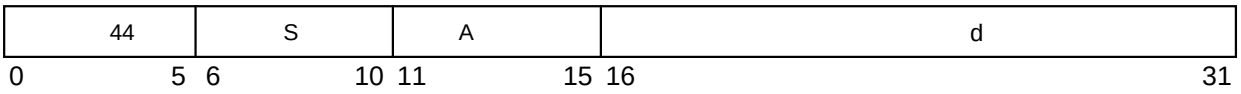
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

sth sth

Store Half Word (x'B000 0000')

sth **r**S,d**(r**A**)**




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 2) ← rS[16-31]
```

EA is the sum (**r**A|0) + d. The contents of the low-order 16 bits of **r**S are stored into the half word in memory addressed by EA.

Other registers altered:

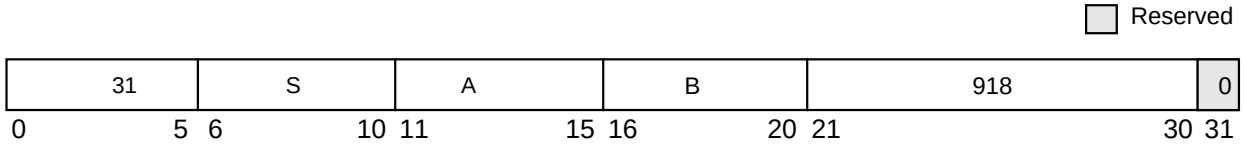
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

sthbrx sthbrx

Store Half Word Byte-Reverse Indexed (x'7C00 072C')

sthbrx **r**S,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA, 2) ← rS[24-31] || rS[16-23]
```

EA is the sum (**r**A|0) + (**r**B). The contents of the low-order eight bits (24-31) of **r**S are stored into bits 0–7 of the half word in memory addressed by EA. The contents of the subsequent low-order eight bits (16-23) of **r**S are stored into bits 8–15 of the half word in memory addressed by EA.

Other registers altered:

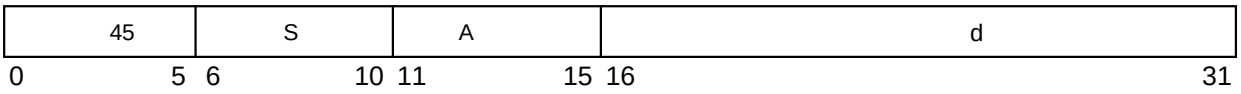
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

sthu sthu

Store Half Word with Update (x'B400 0000')

sthu **r**S,d**(r**A**)**




```
EA ← (rA) + EXTS(d)
MEM(EA, 2) ← rS[16-31]
rA ← EA
```

EA is the sum (**r**A) + d. The contents of the low-order 16 bits of **r**S are stored into the half word in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

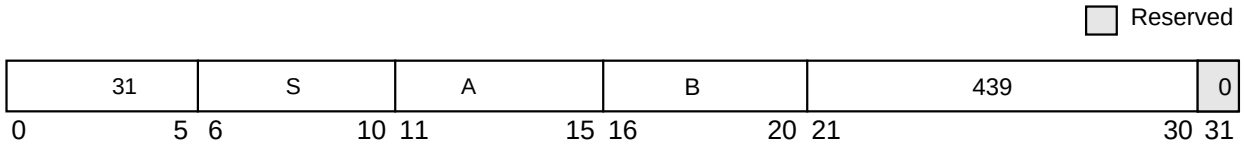
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

sthux sthux

Store Half Word with Update Indexed (x'7C00 036E')

sthux **r**S,**r**A,**r**B




```
EA ← (rA) + (rB)
MEM(EA, 2) ← rS[16-31]
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The contents of the low-order 16 bits of **r**S are stored into the half word in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

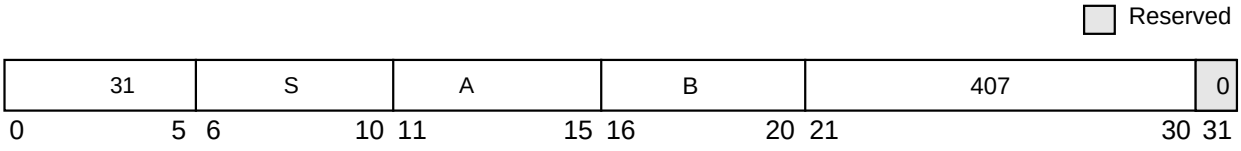
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

sthx sthx

Store Half Word Indexed (x'7C00 032E')

sthx **r**S,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA, 2) ← rS[16-31]
```

EA is the sum (**r**A|0) + (**r**B). The contents of the low-order 16 bits of **r**S are stored into the half word in memory addressed by EA.

Other registers altered:

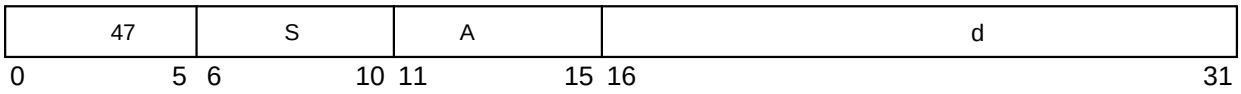
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

stmw stmw

Store Multiple Word (x'BC00 0000')

stmw **r**S,d**(r**A**)**




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
r ← rS
§do while r  ≤ 31
      MEM(EA, 4) ← GPR(r)
      r ← r + 1
      EA ← EA + 4
```

EA is the sum (**r**A|0) + d.

n = (32 – **r**S).

n consecutive words starting at EA are stored from the GPRs **r**S through **r31**. For example, if **r**S = 30, 2 words are stored.

EA must be a multiple of four. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI exceptions, see Section 6.4.3, "DSI Exception (0x00300)," in the *PowerPC Microprocessor Family: The Programming Environments* manual..

NOTE: In some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual store instructions that produce the same results.

Other registers altered:

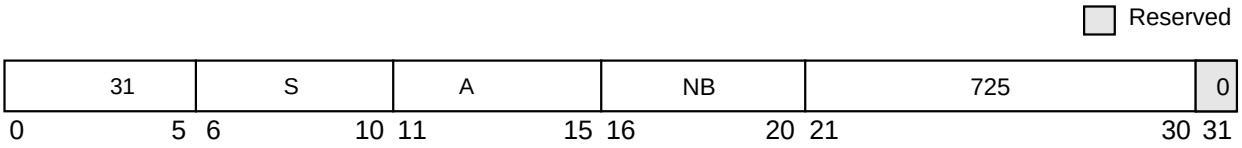
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

stswi stswi

Store String Word Immediate (x'7C00 05AA')

stswi **r**S,**r**A,NB




```
if rA = 0
then EA ← 0
else EA ← (rA)
if NB = 0
then n ← 32
else n ← NB
r ← rS – 1
i ← 0
do while n > 0
     if i = 0
        then r ← r + 1 (mod 32)
     MEM(EA, 1) ← GPR(r)[i,i + 7]
     i ← i + 8
     if i = 32
        then i ← 0
     EA ← EA + 1
     n ← n – 1
```

EA is (**r**A|0). Let *n* = NB if NB not_equal0, *n* = 32 if NB = 0; *n* is the number of bytes to store. Let *nr* = CEIL(*n* / 4);*nr* is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs **r**S through **r**S + *nr* – 1 Bytes are stored left to right from each register. The sequence of registers wraps around through **r0** if required.

Under certain conditions (for example, segment boundary crossing) the data alignment exception handler may be invoked. For additional information about data alignment exceptions, see Section 6.4.3, "DSI Exception (0x00300)," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

NOTE: In some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual store instructions that produce the same results.

Other registers altered:

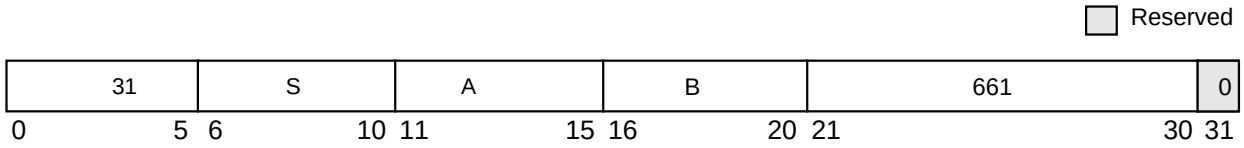
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

stswx stswx

Store String Word Indexed (x'7C00 052A')

stswx **r**S,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
n ← XER[25–31]
r ← rS – 1
i ← 0
do while n > 0
     if i = 0
        then r ← r + 1 (mod 32)
     MEM(EA, 1) ← GPR(r)[i,i + 7]
     i ← i + 8
     if i = 32
        then i ← 0
     EA ← EA + 1
     n ← n – 1
```

EA is the sum (**r**A|0) + (**r**B). Let *n* = XER[25–31]; *n* is the number of bytes to store. Let *nr* = CEIL(*n* / 4);*nr* is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs **r**S through **r**S + *nr* – 1. Bytes are stored left to right from each register. The sequence of registers wraps around through **r0** if required. If *n* = 0, no bytes are stored.

Under certain conditions (for example, segment boundary crossing) the data alignment exception handler may be invoked. For additional information about data alignment exceptions, see Section 6.4.3, "DSI Exception (0x00300)," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

NOTE: In some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual store instructions that produce the same results.

Other registers altered:

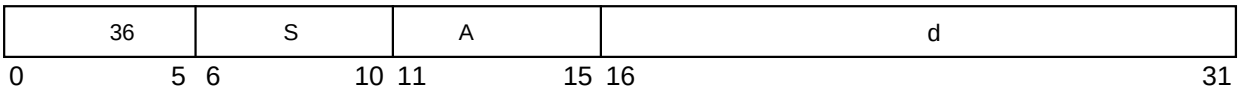
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

stw stw

Store Word (x'9000 0000')

stw **r**S,d(**r**A)




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 4) ← rS
```

EA is the sum (**r**A|0) + d. The contents of **r**S are stored into the word in memory addressed by EA.

Other registers altered:

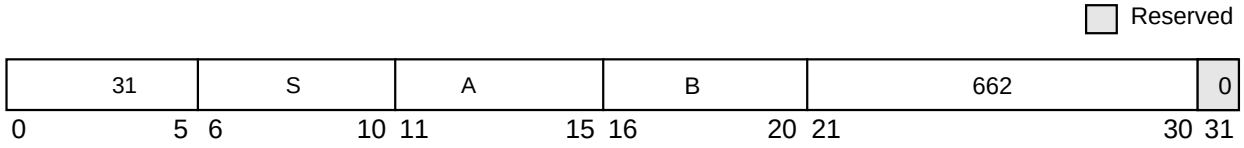
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

stwbrx stwbrx

Store Word Byte-Reverse Indexed (x'7C00 052C')

stwbrx **r**S,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← rS[24-31] || rS[16-23] || rS[8-15] || rS[0-7]
```

EA is the sum (**r**A|0) + (**r**B). The contents of the low-order eight bits (24-31) of **r**S are stored into bits 0–7 of the word in memory addressed by EA. The contents of the subsequent eight low-order bits (16-23) of **r**S are stored into bits 8–15 of the word in memory addressed by EA. The contents of the subsequent eight low-order bits (8-15) of **r**S are stored into bits 16–23 of the word in memory addressed by EA. The contents of the subsequent eight low-order bits (0-7) of **r**S are stored into bits 24–31 of the word in memory addressed by EA.

Other registers altered:

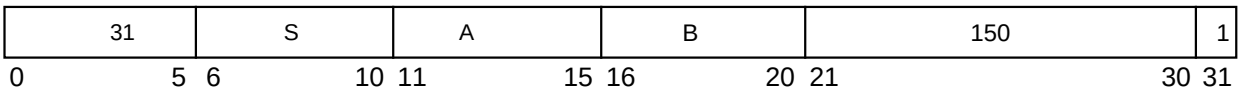
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

stwcx. stwcx.

Store Word Conditional Indexed (x'7C00 012D')

stwcx. **r**S,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
if RESERVE
      then
         MEM(EA, 4) ← rS
         CR0 ← 0b00 || 0b1 || XER[SO]
         RESERVE ← 0
      else
         CR0 ← 0b00 || 0b0 || XER[SO]
```

EA is the sum (**r**A|0) + (**r**B). If the reserved bit is set, the **stwcx.** instruction stores **r**S to effective address (**r**A + **r**B), clears the reserved bit, and sets CR0[EQ]. If the reserved bit is not set, the **stwcx.** instruction does not do a store; it leaves the reserved bit cleared and clears CR0[EQ]. Software must look at CR0[EQ] to see if the **stwcx.** was successful.

The reserved bit is set by the **lwarx** instruction. The reserved bit is cleared by any **stwcx.** instruction to any address, and also by snooping logic if it detects that another processor does any kind of write or invalidate to the block indicated in the reservation buffer when reserved is set.

EA must be a multiple of four. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI exceptions, see Section 6.4.3, "DSI Exception (0x00300)," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

The granularity with which reservations are managed is implementation-dependent. Therefore, the memory to be accessed by the load and reserve and store conditional instructions should be controlled by a system library program.

Because the hardware doesn't compare reservation address when executing the stwcx. instruction, operating systems software MUST reset the reservation if an exception or other type of interrupt occurs to insure atomic memory references of **lwarx** and **stwcx.** pairs.

Other registers altered:

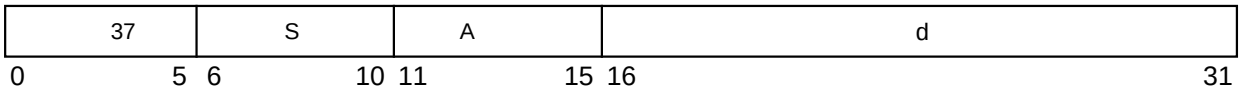
- CR0 field is set to reflect whether the store operation was performed as follows:
  CR0[LT GT EQ S0] = 0b00 || store_performed || XER[SO]
- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

stwu stwu

Store Word with Update (x'9400 0000')

stwu **r**S,d**(r**A**)**




```
EA ← (rA) + EXTS(d)
MEM(EA, 4) ← rS
rA ← EA
```

EA is the sum (**r**A) + d. The contents of **r**S are stored into the word in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

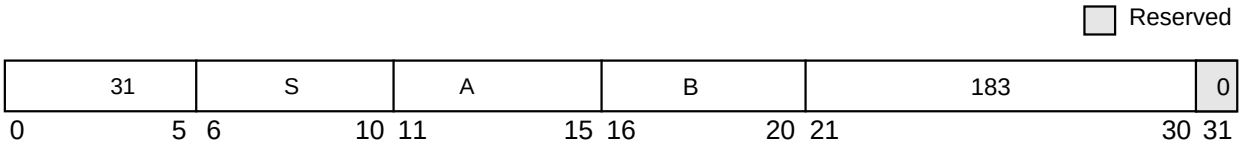
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

stwux stwux

Store Word with Update Indexed (x'7C00 016E')

stwux **r**S,**r**A,**r**B




```
EA ← (rA) + (rB)
MEM(EA, 4) ← rS
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The contents of **r**S are stored into the word in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

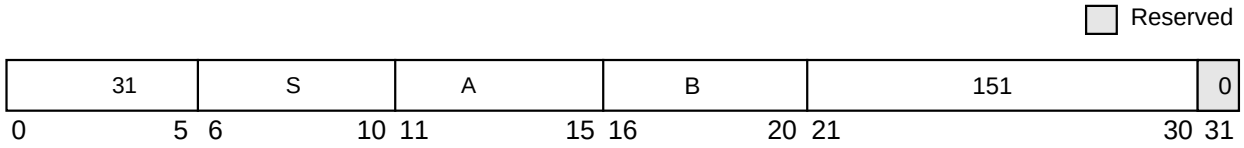
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

stwx stwx

Store Word Indexed (x'7C00 012E')

stwx **r**S,**r**A,**r**B




```
if rA = 0
then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← rS
```

EA is the sum (**r**A|0) + (**r**B). The contents of **r**S are stored into the word in memory addressed by EA.

Other registers altered:

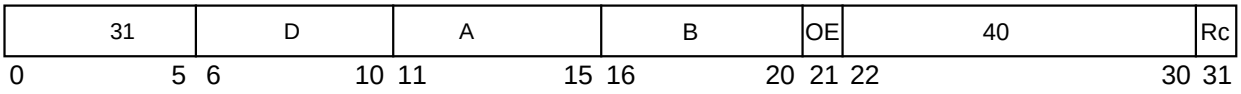
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

subf*x* subf*x*

Subtract From (x'7C00 0050')

subf	**r**D,**r**A,**r**B	(OE = 0 Rc = 0)
subf.	**r**D,**r**A,**r**B	(OE = 0 Rc = 1)
subfo	**r**D,**r**A,**r**B	(OE = 1 Rc = 0)
subfo.	**r**D,**r**A,**r**B	(OE = 1 Rc = 1)




```
rD ← ¬ (rA) + (rB) + 1
```

The sum ¬ (**r**A) + (**r**B) + 1 is placed into **r**D. (equivlent to (rB)--(**r**A))

The **subf** instruction is preferred for subtraction because it sets few status bits.

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
- XER:
  Affected: SO, OV (if OE = 1)

Simplified mnemonics:

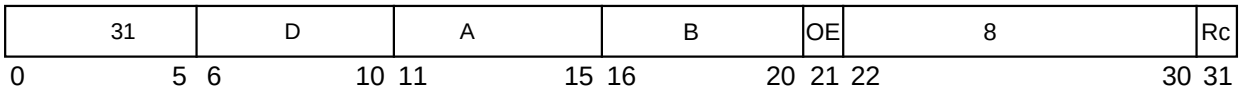
sub **r**D,**r**A,**r**B equivalent to **subf** **r**D,**r**B,**r**A

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

subfc*x* subfc*x*

Subtract from Carrying (x’7C00 0010’)

subfc	**r**D,**r**A,**r**B	(OE = 0 Rc = 0)
subfc.	**r**D,**r**A,**r**B	(OE = 0 Rc = 1)
subfco	**r**D,**r**A,**r**B	(OE = 1 Rc = 0)
subfco.	**r**D,**r**A,**r**B	(OE = 1 Rc = 1)




```
rD ← ¬ (rA) + (rB) + 1
```

The sum ¬ (**r**A) + (**r**B) + 1 is placed into **r**D. (equivlent to (rB)--(**r**A))

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

  NOTE: CR0 field may not reflect the infinitely precise result if overflow occurs (see next).

- XER:

  Affected: CA

  Affected: SO, OV (if OE = 1)

NOTE: The setting of the affected bits in the XER reflects overflow of the 32-bit results. For further information see Chapter 3, “Operand Conventions” in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Simplified mnemonics:

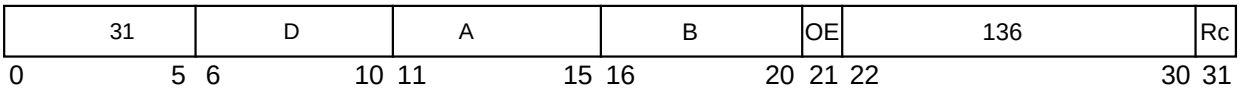
subc **r**D,**r**A,**r**B equivalent to **subfc** **r**D,**r**B,**r**A

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

subfe*x* subfe*x*

Subtract from Extended (x'7C00 0110')

subfe **r**D,**r**A,**r**B (OE = 0 Rc = 0)
subfe. **r**D,**r**A,**r**B (OE = 0 Rc = 1)
subfeo **r**D,**r**A,**r**B (OE = 1 Rc = 1)
subfeo. **r**D,**r**A,**r**B (OE = 1 Rc = 1)




```
rD ← ¬ (rA) + (rB) + XER[CA]
```

The sum ¬ (**r**A) + (**r**B) + XER[CA] is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

  NOTE: CR0 field may not reflect the infinitely precise result if overflow occurs (See Chapter 3, "Operand Conventions" in the *PowerPC Microprocessor Family: The Programming Environments* manual for setting of affected bits.)

- XER:

  Affected: CA

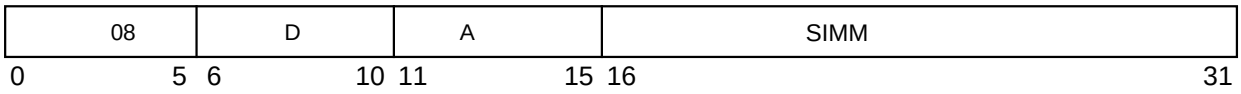
  Affected: SO, OV (if OE = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

subfic subfic

Subtract from Immediate Carrying (x’2000 0000’)

subfic **r**D,**r**A,SIMM




```
rD ← ¬ (rA) + EXTS(SIMM) + 1
```

The sum ¬ (**r**A) + EXTS(SIMM) + 1 is placed into **r**D (Equivalent to EXTS(SIMM)-(**r**A)).

Other registers altered:

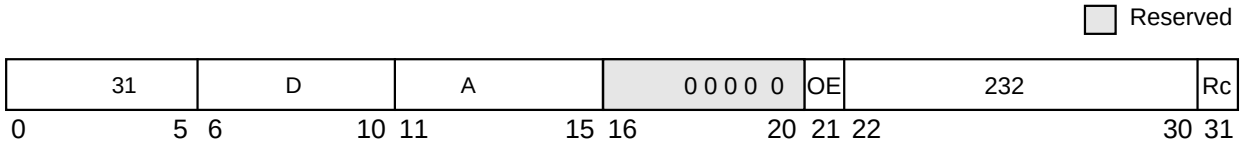
- XER:
  Affected: CA

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

subfme*x* subfme*x*

Subtract from Minus One Extended (x’7C00 01D0’)

subfme	**r**D,**r**A	(OE = 0 Rc = 0)
subfme.	**r**D,**r**A	(OE = 0 Rc = 1)
subfmeo	**r**D,**r**A	(OE = 1 Rc = 0)
subfmeo.	**r**D,**r**A	(OE = 1 Rc = 1)




```
rD ← ¬ (rA) + XER[CA] – 1
```

The sum ¬ (**r**A) + XER[CA] + (32)1 is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

  NOTE: CR0 field may not reflect the infinitely precise result if overflow occurs (See Chapter 3, “Operand Conventions,” in the *PowerPC Microprocessor Family: The Programming Environments* manual.)

- XER:

  Affected: CA

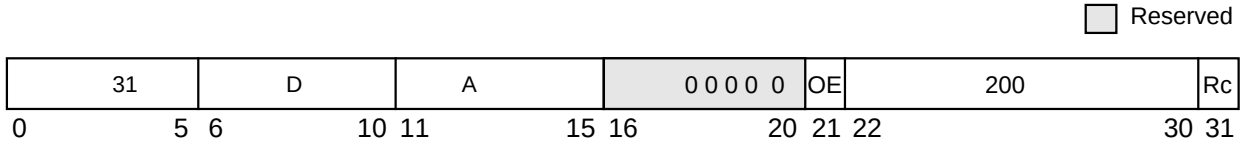
  Affected: SO, OV (if OE = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

subfze*x* subfze*x*

Subtract from Zero Extended (x'7C00 0190')

subfze	**r**D,**r**A	(OE = 0 Rc = 0)
subfze.	**r**D,**r**A	(OE = 0 Rc = 1)
subfzeo	**r**D,**r**A	(OE = 1 Rc = 0)
subfzeo.	**r**D,**r**A	(OE = 1 Rc = 1)




```
rD ← ¬ (rA) + XER[CA]
```

The sum ¬ (**r**A) + XER[CA] is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

  NOTE: CR0 field may not reflect the infinitely precise result if overflow occurs (see next).

- XER:

  Affected: CA

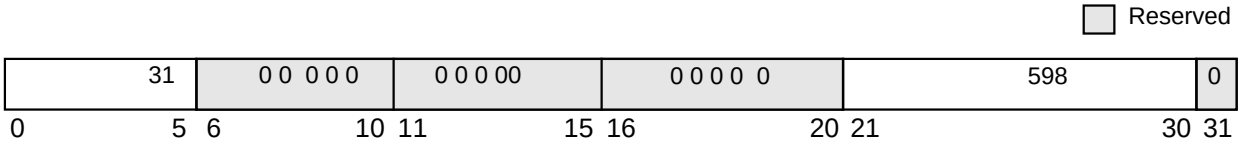
  Affected: SO, OV (if OE = 1)

  NOTE: See Chapter 3, "Operand Conventions," in the *PowerPC Microprocessor Family: The Programming Environments* manual.

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				XO

sync sync

Synchronize (x'7C00 04AC')



The **sync** instruction provides an ordering function for the effects of all instructions executed by a given processor. Executing a **sync** instruction ensures that all instructions preceding the **sync** instruction appear to have completed before the **sync** instruction completes, and that no subsequent instructions are initiated by the processor until after the **sync** instruction completes. When the **sync** instruction completes, all external accesses caused by instructions preceding the **sync** instruction will have been performed with respect to all other mechanisms that access memory. For more information on how the **sync** instruction affects the VEA, refer to Chapter 5, "Cache Model and Memory Coherency" in the *PowerPC Microprocessor Family: The Programming Environments* manual. Multiprocessor implementations also send a **sync** address-only broadcast that is useful in some designs. For example, if a design has an external buffer that re-orders loads and stores for better bus efficiency, the **sync** broadcast signals to that buffer that previous loads/stores must be completed before any following loads/stores.

The **sync** instruction can be used to ensure that the results of all stores into a data structure, caused by store instructions executed in a "critical section" of a program, are seen by other processors before the data structure is seen as unlocked.

The functions performed by the **sync** instruction will normally take a significant amount of time to complete, so indiscriminate use of this instruction may adversely affect performance. In addition, the time required to execute **sync** may vary from one execution to another.

The **eieio** instruction may be more appropriate than **sync** for many cases.

This instruction is execution synchronizing. For more information on execution synchronization, see Section 4.1.5, "Synchronizing Instructions," in *The Programming Environments Manual.*

Other registers altered:

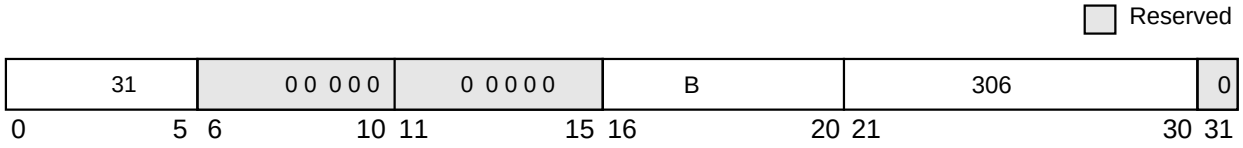
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

tlbie tlbie

Translation Lookaside Buffer Invalidate Entry (x'7C00 0264')

tlbie **r**B




```
VPS ← rB[4-19]
Identify TLB entries corresponding to VPS
Each such TLB entry ← invalid
```

EA is the contents of **r**B. If the translation lookaside buffer (TLB) contains an entry corresponding to EA, that entry is made invalid (that is, removed from the TLB).

Multiprocessing implementations (for example, the 601, and 604) send a **tlbie** address-only broadcast over the address bus to tell other processors to invalidate the same TLB entry in their TLBs.

The TLB search is done regardless of the settings of MSR[IR] and MSR[DR]. The search is done based on a portion of the logical page number within a segment, without reference to the SLB, segment table, or segment registers. All entries matching the search criteria are invalidated.

Block address translation for EA, if any, is ignored. Refer to Section 7.5.3.4, "Synchronization of Memory Accesses and Referenced and Changed Bit Updates" and Section 7.6.3, "Page Table Updates" in the *PowerPC Microprocessor Family: The Programming Environments* manual for other requirements associated with the use of this instruction.

This is a supervisor-level instruction and optional in the PowerPC architecture.

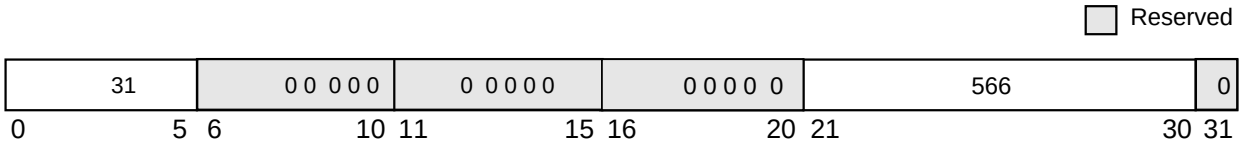
Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
OEA	YES		YES	X

tlbsync tlbsync

TLB Synchronize (x'7C00 046C')



If an implementation sends a broadcast for **tlbie** then it will also send a broadcast for **tlbsync**. Executing a **tlbsync** instruction ensures that all **tlbie** instructions previously executed by the processor executing the **tlbsync** instruction have completed on all other processors.

The operation performed by this instruction is treated as a caching-inhibited and guarded data access with respect to the ordering done by **eieio**.

NOTE: The 601 expands the use of the **sync** instruction to cover **tlbsync** functionality.

Refer to Section 7.5.3.4, "Synchronization of Memory Accesses and Referenced and Changed Bit Updates" and Section 7.6.3, "Page Table Updates" in the *PowerPC Microprocessor Family: The Programming Environments* manual for other requirements associated with the use of this instruction.

This instruction is supervisor-level and optional in the PowerPC architecture.

Other registers altered:

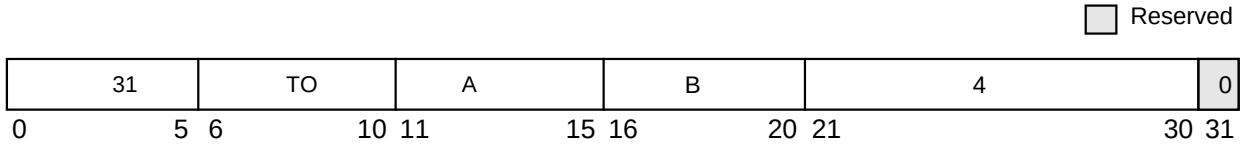
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
OEA	YES		YES	X

tw tw

Trap Word (x’7C00 0008’)

tw TO,**r**A,**r**B




```
a ← EXTS(rA)
b ← EXTS(rB)
if (a < b) & TO[0] then TRAP
if (a > b) & TO[1] then TRAP
if (a = b) & TO[2] then TRAP
if (a <U b) & TO[3] then TRAP
if (a >U b) & TO[4] then TRAP
```

The contents of **r**A are compared arithmetically with the contents **r**B for TO[0, 1, 2]. The contents of **r**A are compared logically with the contents **r**B for TO[3, 4]. If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Other registers altered:

- None

Simplified mnemonics:

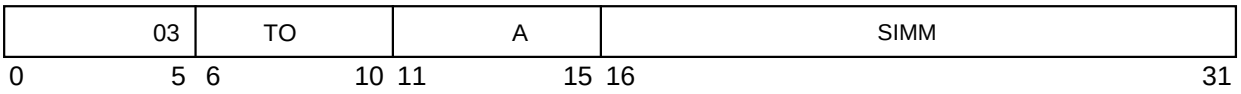
tweq rA,**r**B	equivalent to	**tw**	**4,r**A,**r**B
twlgerA,**r**B	equivalent to	**tw**	**5,r**A,**r**B
trap	equivalent to	**tw**	**31,0,0**

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

twi twi

Trap Word Immediate (x'0C00 0000')

twi TO,**r**A,SIMM




```
a ← EXTS(rA)
if (a < EXTS(SIMM)) & TO[0] then TRAP
if (a > EXTS(SIMM)) & TO[1] then TRAP
if (a = EXTS(SIMM)) & TO[2] then TRAP
if (a <U EXTS(SIMM)) & TO[3] then TRAP
if (a >U EXTS(SIMM)) & TO[4] then TRAP
```

The contents of **r**A are compared arithmetically with the sign-extended value of the SIMM field for TO[0, 1, 2]. The contents of **r**A are compared logically with the sign-extended value of the SIMM field for TO[3, 4]. If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Other registers altered:

- None

Simplified mnemonics:

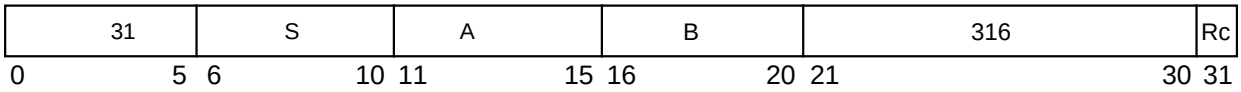
twgti **r**A,value	equivalent to	**twi**	**8,r**A,value
twllei **r**A,value	equivalent to	**twi**	**6,r**A,value

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

xor*x* xor*x*

XOR (x'7C00 0278')

xor **r**A,**r**S,**r**B (Rc = 0)
xor. **r**A,**r**S,**r**B (Rc = 1)



rA ← (**r**S) ⊕ (**r**B)

The contents of **r**S are XORed with the contents of **r**B and the result is placed into **r**A.

Other registers altered:

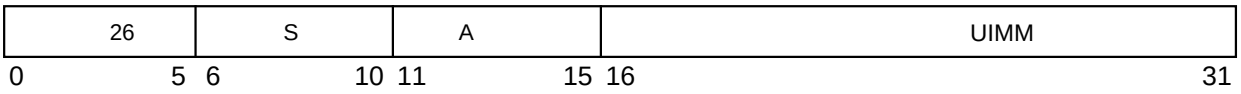
- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				X

xori xori

XOR Immediate (x'6800 0000')

xori **r**A,**r**S,UIMM




```
rA ← (rS) ⊕ ((16)0 || UIMM)
```

The contents of **r**S are XORed with 0x0000 || UIMM and the result is placed into **r**A.

Other registers altered:

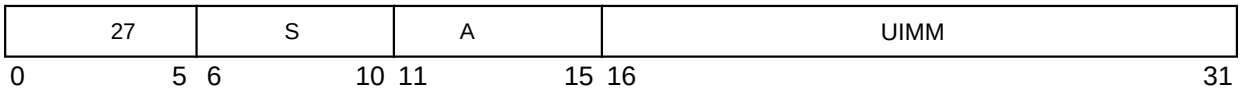
- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

xoris xoris

XOR Immediate Shifted (x'6C00 0000')

xoris **r**A,**r**S,UIMM



rA ← (**r**S) ⊕ (UIMM || (16)0)

The contents of **r**S are XORed with UIMM || 0x0000 and the result is placed into **r**A.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Gekko Specific	PowerPC Optional	Form
UISA				D

Appendix A– Gekko Instruction Set

A.1 Instructions Sorted by Opcode

Table A-1 lists the instructions defined in the PowerPC architecture in numeric order by opcode.

Key:

Reserved bits

Table A-1 Complete Instruction List Sorted by Opcode

Name	0 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
twi	0 0 0 0 1 1	TO		A	SIMM			
ps_cmpu0	0 0 0 1 0 0	crfD	0 0	A	B	0 0 0 0 0 0 0 0 0 0		0
psq_lx	0 0 0 1 0 0	D		A	B	w i	0 0 0 1 1 0	0
psq_stx	0 0 0 1 0 0	S		A	B	w i	0 0 0 1 1 1	0
ps_sum0	0 0 0 1 0 0	D		A	B	C	0 1 0 1 0	Rc
ps_sum1	0 0 0 1 0 0	D		A	B	C	0 1 0 1 1	Rc
ps_muls0	0 0 0 1 0 0	D		A	0 0 0 0 0	C	0 1 1 0 0	Rc
ps_muls1	0 0 0 1 0 0	D		A	0 0 0 0 0	C	0 1 1 0 1	Rc
ps_madds0	0 0 0 1 0 0	D		A	B	C	0 1 1 1 0	Rc
ps_madds1	0 0 0 1 0 0	D		A	B	C	0 1 1 1 1	Rc
ps_div	0 0 0 1 0 0	D		A	B	0 0 0 0 0	1 0 0 1 0	Rc
ps_sub	0 0 0 1 0 0	D		A	B	0 0 0 0 0	1 0 1 0 0	Rc
ps_add	0 0 0 1 0 0	D		A	B	0 0 0 0 0	1 0 1 0 1	Rc
ps_sel	0 0 0 1 0 0	D		A	B	C	1 0 1 1 1	Rc
ps_res	0 0 0 1 0 0	D		00000	B	00000	1 1 0 0 0	Rc
ps_mul	0 0 0 1 0 0	D		A	00000	C	1 1 0 0 1	Rc
ps_rsqrte	0 0 0 1 0 0	D		00000	B	00000	1 1 0 1 0	Rc
ps_msub	0 0 0 1 0 0	D		A	B	C	1 1 1 0 0	Rc
ps_madd	0 0 0 1 0 0	D		A	B	C	1 1 1 0 1	Rc
ps_nmsub	0 0 0 1 0 0	D		A	B	C	1 1 1 1 0	Rc

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ps_nmadd	0 0 0 1 0 0		D					A					B					C					1	1	1	1	1	Rc
ps_cmpo0	0 0 0 1 0 0		crfD			0	0	A					B					0	0	0	0	1	0	0	0	0	0	0
psq_lux	0 0 0 1 0 0		D					A					B					w	i				1	0	0	1	1	0
psq_stux	0 0 0 1 0 0		S					A					B					w	i				1	0	0	1	1	1
ps_neg	0 0 0 1 0 0		D					0	0	0	0	0	B					0	0	0	0	1	0	1	0	0	0	Rc
ps_cmpu1	0 0 0 1 0 0		crfD			0	0	A					B					0	0	0	1	0	0	0	0	0	0	0
ps_mr	0 0 0 1 0 0		D					0	0	0	0	0	B					0	0	0	1	0	0	1	0	0	0	Rc
ps_cmpo1	0 0 0 1 0 0		crfD			0	0	A					B					0	0	0	1	1	0	0	0	0	0	0
ps_nabs	0 0 0 1 0 0		D					0	0	0	0	0	B					0	0	1	0	0	0	1	0	0	0	Rc
ps_abs	0 0 0 1 0 0		D					0	0	0	0	0	B					0	1	0	0	0	0	1	0	0	0	Rc
ps_merge00	0 0 0 1 0 0		D					A					B					1	0	0	0	0	1	0	0	0	0	Rc
ps_merge01	0 0 0 1 0 0		D					A					B					1	0	0	0	1	1	0	0	0	0	Rc
ps_merge10	0 0 0 1 0 0		D					A					B					1	0	0	1	0	1	0	1	0	1	Rc
ps_merge11	0 0 0 1 0 0		D					A					B					1	0	0	1	1	1	0	0	0	0	Rc
dcbz_l	0 0 0 1 0 0		0	0	0	0	0	A					B					1	1	1	1	1	1	0	1	1	0	0
mulli	0 0 0 1 1 1		D					A					SIMM															
subfic	0 0 1 0 0 0		D					A					SIMM															
cmpli	0 0 1 0 1 0		crfD			0	L	A					UIMM															
cmpi	0 0 1 0 1 1		crfD			0	L	A					SIMM															
addic	0 0 1 1 0 0		D					A					SIMM															
addic.	0 0 1 1 0 1		D					A					SIMM															
addi	0 0 1 1 1 0		D					A					SIMM															
addis	0 0 1 1 1 1		D					A					SIMM															
bc*x*	0 1 0 0 0 0		BO					BI					BD													AA	LK
sc	0 1 0 0 0 1		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
b*x*	0 1 0 0 1 0		LI																								AA	LK
mcrf	0 1 0 0 1 1		crfD			0	0	crfS			0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
bclr*x*	0 1 0 0 1 1		BO					BI					0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	LK
crnor	0 1 0 0 1 1		crbD					crbA					crbB					0	0	0	0	1	0	0	0	0	1	0
rfi	0 1 0 0 1 1		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0
crandc	0 1 0 0 1 1		crbD					crbA					crbB					0	0	1	0	0	0	0	0	0	1	0
isync	0 1 0 0 1 1		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	0
crxor	0 1 0 0 1 1		crbD					crbA					crbB					0	0	1	1	0	0	0	0	0	1	0

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
crnand	0 1 0 0 1 1	crbD	crbA	crbB	0 0 1 1 1 0 0 0 0 1		0
crand	0 1 0 0 1 1	crbD	crbA	crbB	0 1 0 0 0 0 0 0 0 1		0
creqv	0 1 0 0 1 1	crbD	crbA	crbB	0 1 0 0 1 0 0 0 0 1		0
crorc	0 1 0 0 1 1	crbD	crbA	crbB	0 1 1 0 1 0 0 0 0 1		0
cror	0 1 0 0 1 1	crbD	crbA	crbB	0 1 1 1 0 0 0 0 0 1		0
bcctr*x*	0 1 0 0 1 1	BO	BI	0 0 0 0 0	1 0 0 0 0 1 0 0 0 0		LK
rlwimi*x*	0 1 0 1 0 0	S	A	SH	MB	ME	Rc
rlwinm*x*	0 1 0 1 0 1	S	A	SH	MB	ME	Rc
rlwnm*x*	0 1 0 1 1 1	S	A	B	MB	ME	Rc
ori	0 1 1 0 0 0	S	A	UIMM			
oris	0 1 1 0 0 1	S	A	UIMM			
xori	0 1 1 0 1 0	S	A	UIMM			
xoris	0 1 1 0 1 1	S	A	UIMM			
andi.	0 1 1 1 0 0	S	A	UIMM			
andis.	0 1 1 1 0 1	S	A	UIMM			
cmp	0 1 1 1 1 1	crfD 0 L	A	B	0 0 0 0 0 0 0 0 0 0		0
tw	0 1 1 1 1 1	TO	A	B	0 0 0 0 0 0 0 1 0 0		0
subfc*x*	0 1 1 1 1 1	D	A	B	OE	0 0 0 0 0 0 1 0 0 0	Rc
addc*x*	0 1 1 1 1 1	D	A	B	OE	0 0 0 0 0 0 1 0 1 0	Rc
mulhwu*x*	0 1 1 1 1 1	D	A	B	0	0 0 0 0 0 0 1 0 1 1	Rc
mfcr	0 1 1 1 1 1	D	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 1 0 0 1 1		0
lwarx	0 1 1 1 1 1	D	A	B	0 0 0 0 0 1 0 1 0 0		0
lwzx	0 1 1 1 1 1	D	A	B	0 0 0 0 0 1 0 1 1 1		0
slw*x*	0 1 1 1 1 1	S	A	B	0 0 0 0 0 1 1 0 0 0		Rc
cntlzw*x*	0 1 1 1 1 1	S	A	0 0 0 0 0	0 0 0 0 0 1 1 0 1 0		Rc
and*x*	0 1 1 1 1 1	S	A	B	0 0 0 0 0 1 1 1 0 0		Rc
cmpl	0 1 1 1 1 1	crfD 0 L	A	B	0 0 0 0 1 0 0 0 0 0		0
subf*x*	0 1 1 1 1 1	D	A	B	OE	0 0 0 0 1 0 1 0 0 0	Rc
dcbst	0 1 1 1 1 1	0 0 0 0 0	A	B	0 0 0 0 1 1 0 1 1 0		0
lwzux	0 1 1 1 1 1	D	A	B	0 0 0 0 1 1 0 1 1 1		0
andc*x*	0 1 1 1 1 1	S	A	B	0 0 0 0 1 1 1 1 0 0		Rc
mulhw*x*	0 1 1 1 1 1	D	A	B	0	0 0 0 1 0 0 1 0 1 1	Rc
mfmsr	0 1 1 1 1 1	D	0 0 0 0 0	0 0 0 0 0	0 0 0 1 0 1 0 0 1 1		0

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
dcbf	0 1 1 1 1 1		0 0 0 0 0					A					B					0 0 0 1 0 1 0 1 1 0										0
lbzx	0 1 1 1 1 1		D					A					B					0 0 0 1 0 1 0 1 1 1										0
neg*x*	0 1 1 1 1 1		D					A					0 0 0 0 0					OE	0 0 0 1 1 0 1 0 0 0										Rc
lbzux	0 1 1 1 1 1		D					A					B					0 0 0 1 1 1 0 1 1 1										0
nor*x*	0 1 1 1 1 1		S					A					B					0 0 0 1 1 1 1 1 0 0										Rc
subfe*x*	0 1 1 1 1 1		D					A					B					OE	0 0 1 0 0 0 1 0 0 0										Rc
adde*x*	0 1 1 1 1 1		D					A					B					OE	0 0 1 0 0 0 1 0 1 0										Rc
mtcrf	0 1 1 1 1 1		S					0	CRM								0	0 0 1 0 0 1 0 0 0 0										0
mtmsr	0 1 1 1 1 1		S					0 0 0 0 0					0 0 0 0 0					0 0 1 0 0 1 0 0 1 0										0
stwcx.	0 1 1 1 1 1		S					A					B					0 0 1 0 0 1 0 1 1 0										1
stwx	0 1 1 1 1 1		S					A					B					0 0 1 0 0 1 0 1 1 1										0
stwux	0 1 1 1 1 1		S					A					B					0 0 1 0 1 1 0 1 1 1										0
subfze*x*	0 1 1 1 1 1		D					A					0 0 0 0 0					OE	0 0 1 1 0 0 1 0 0 0										Rc
addze*x*	0 1 1 1 1 1		D					A					0 0 0 0 0					OE	0 0 1 1 0 0 1 0 1 0										Rc
mtsr	0 1 1 1 1 1		S					0	SR				0 0 0 0 0					0 0 1 1 0 1 0 0 1 0										0
stbx	0 1 1 1 1 1		S					A					B					0 0 1 1 0 1 0 1 1 1										0
subfme*x*	0 1 1 1 1 1		D					A					0 0 0 0 0					OE	0 0 1 1 1 0 1 0 0 0										Rc
addme*x*	0 1 1 1 1 1		D					A					0 0 0 0 0					OE	0 0 1 1 1 0 1 0 1 0										Rc
mullw*x*	0 1 1 1 1 1		D					A					B					OE	0 0 1 1 1 0 1 0 1 1										Rc
mtsrin	0 1 1 1 1 1		S					0 0 0 0 0					B					0 0 1 1 1 1 0 0 1 0										0
dcbtst	0 1 1 1 1 1		0 0 0 0 0					A					B					0 0 1 1 1 1 0 1 1 0										0
stbux	0 1 1 1 1 1		S					A					B					0 0 1 1 1 1 0 1 1 1										0
add*x*	0 1 1 1 1 1		D					A					B					OE	0 1 0 0 0 0 1 0 1 0										Rc
dcbt	0 1 1 1 1 1		0 0 0 0 0					A					B					0 1 0 0 0 1 0 1 1 0										0
lhzx	0 1 1 1 1 1		D					A					B					0 1 0 0 0 1 0 1 1 1										0
eqv*x*	0 1 1 1 1 1		S					A					B					0 1 0 0 0 1 1 1 0 0										Rc
tlbie	0 1 1 1 1 1		0 0 0 0 0					0 0 0 0 0					B					0 1 0 0 1 1 0 0 1 0										0
eciwx	0 1 1 1 1 1		D					A					B					0 1 0 0 1 1 0 1 1 0										0
lhzux	0 1 1 1 1 1		D					A					B					0 1 0 0 1 1 0 1 1 1										0
xor*x*	0 1 1 1 1 1		S					A					B					0 1 0 0 1 1 1 1 0 0										Rc
mfspr	0 1 1 1 1 1		D					spr										0 1 0 1 0 1 0 0 1 1										0
lhax	0 1 1 1 1 1		D					A					B					0 1 0 1 0 1 0 1 1 1										0
mftb	0 1 1 1 1 1		D					tbr										0 1 0 1 1 1 0 0 1 1										0

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
lhaux	0 1 1 1 1 1	D	A	B	0 1 0 1 1 1 0 1 1 1	0
sthx	0 1 1 1 1 1	S	A	B	0 1 1 0 0 1 0 1 1 1	0
orc*x*	0 1 1 1 1 1	S	A	B	0 1 1 0 0 1 1 1 0 0	Rc
ecowx	0 1 1 1 1 1	S	A	B	0 1 1 0 1 1 0 1 1 0	0
sthux	0 1 1 1 1 1	S	A	B	0 1 1 0 1 1 0 1 1 1	0
or*x*	0 1 1 1 1 1	S	A	B	0 1 1 0 1 1 1 1 0 0	Rc
divwu*x*	0 1 1 1 1 1	D	A	B	OE 0 1 1 1 0 0 1 0 1 1	Rc
mtspr	0 1 1 1 1 1	S	spr		0 1 1 1 0 1 0 0 1 1	0
dcbi	0 1 1 1 1 1	0 0 0 0 0	A	B	0 1 1 1 0 1 0 1 1 0	0
nand*x*	0 1 1 1 1 1	S	A	B	0 1 1 1 0 1 1 1 0 0	Rc
divw*x*	0 1 1 1 1 1	D	A	B	OE 0 1 1 1 1 0 1 0 1 1	Rc
mcrxr	0 1 1 1 1 1	crfD 0 0	0 0 0 0 0	0 0 0 0 0	1 0 0 0 0 0 0 0 0 0	0
lswx	0 1 1 1 1 1	D	A	B	1 0 0 0 0 1 0 1 0 1	0
lwbrx	0 1 1 1 1 1	D	A	B	1 0 0 0 0 1 0 1 1 0	0
lfsx	0 1 1 1 1 1	D	A	B	1 0 0 0 0 1 0 1 1 1	0
srw*x*	0 1 1 1 1 1	S	A	B	1 0 0 0 0 1 1 0 0 0	Rc
tlbsync	0 1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	1 0 0 0 1 1 0 1 1 0	0
lfsux	0 1 1 1 1 1	D	A	B	1 0 0 0 1 1 0 1 1 1	0
mfsr	0 1 1 1 1 1	D	0 SR	0 0 0 0 0	1 0 0 1 0 1 0 0 1 1	0
lswi	0 1 1 1 1 1	D	A	NB	1 0 0 1 0 1 0 1 0 1	0
sync	0 1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	1 0 0 1 0 1 0 1 1 0	0
lfdx	0 1 1 1 1 1	D	A	B	1 0 0 1 0 1 0 1 1 1	0
lfdux	0 1 1 1 1 1	D	A	B	1 0 0 1 1 1 0 1 1 1	0
mfsrin	0 1 1 1 1 1	D	0 0 0 0 0	B	1 0 1 0 0 1 0 0 1 1	0
stswx	0 1 1 1 1 1	S	A	B	1 0 1 0 0 1 0 1 0 1	0
stwbrx	0 1 1 1 1 1	S	A	B	1 0 1 0 0 1 0 1 1 0	0
stfsx	0 1 1 1 1 1	S	A	B	1 0 1 0 0 1 0 1 1 1	0
stfsux	0 1 1 1 1 1	S	A	B	1 0 1 0 1 1 0 1 1 1	0
stswi	0 1 1 1 1 1	S	A	NB	1 0 1 1 0 1 0 1 0 1	0
stfdx	0 1 1 1 1 1	S	A	B	1 0 1 1 0 1 0 1 1 1	0
stfdux	0 1 1 1 1 1	S	A	B	1 0 1 1 1 1 0 1 1 1	0
lhbrx	0 1 1 1 1 1	D	A	B	1 1 0 0 0 1 0 1 1 0	0
sraw*x*	0 1 1 1 1 1	S	A	B	1 1 0 0 0 1 1 0 0 0	Rc

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
srawi*x*	0 1 1 1 1 1		S					A					SH					1 1 0 0 1 1 1 0 0 0									Rc
eieio	0 1 1 1 1 1		0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					1 1 0 1 0 1 0 1 1 0									0
sthbrx	0 1 1 1 1 1		S					A					B					1 1 1 0 0 1 0 1 1 0									0
extsh*x*	0 1 1 1 1 1		S					A					0 0 0 0 0					1 1 1 0 0 1 1 0 1 0									Rc
extsb*x*	0 1 1 1 1 1		S					A					0 0 0 0 0					1 1 1 0 1 1 1 0 1 0									Rc
icbi	0 1 1 1 1 1		0 0 0 0 0					A					B					1 1 1 1 0 1 0 1 1 0									0
stfiwx	0 1 1 1 1 1		S					A					B					1 1 1 1 0 1 0 1 1 1									0
dcbz	0 1 1 1 1 1		0 0 0 0 0					A					B					1 1 1 1 1 1 0 1 1 0									0
lwz	1 0 0 0 0 0		D					A					d														
lwzu	1 0 0 0 0 1		D					A					d														
lbz	1 0 0 0 1 0		D					A					d														
lbzu	1 0 0 0 1 1		D					A					d														
stw	1 0 0 1 0 0		S					A					d														
stwu	1 0 0 1 0 1		S					A					d														
stb	1 0 0 1 1 0		S					A					d														
stbu	1 0 0 1 1 1		S					A					d														
lhz	1 0 1 0 0 0		D					A					d														
lhzu	1 0 1 0 0 1		D					A					d														
lha	1 0 1 0 1 0		D					A					d														
lhau	1 0 1 0 1 1		D					A					d														
sth	1 0 1 1 0 0		S					A					d														
sthu	1 0 1 1 0 1		S					A					d														
lmw	1 0 1 1 1 0		D					A					d														
stmw	1 0 1 1 1 1		S					A					d														
lfs	1 1 0 0 0 0		D					A					d														
lfsu	1 1 0 0 0 1		D					A					d														
lfd	1 1 0 0 1 0		D					A					d														
lfdu	1 1 0 0 1 1		D					A					d														
stfs	1 1 0 1 0 0		S					A					d														
stfsu	1 1 0 1 0 1		S					A					d														
stfd	1 1 0 1 1 0		S					A					d														
stfdu	1 1 0 1 1 1		S					A					d														
psq_l	1 1 1 0 0 0		D					A					w	i				d									

Name	0 5	6 7	8 9	10 11	12 13 14 15	16 17 18 19	20 21 22 23 24 25	26 27 28 29 30	31
psq_lu	1 1 1 0 0 1	D		A	w	i	d		
fdivs*x*	1 1 1 0 1 1	D		A	B		0 0 0 0 0	1 0 0 1 0	Rc
fsubs*x*	1 1 1 0 1 1	D		A	B		0 0 0 0 0	1 0 1 0 0	Rc
fadds*x*	1 1 1 0 1 1	D		A	B		0 0 0 0 0	1 0 1 0 1	Rc
fres*x*	1 1 1 0 1 1	D		0 0 0 0 0	B		0 0 0 0 0	1 1 0 0 0	Rc
fmuls*x*	1 1 1 0 1 1	D		A	0 0 0 0 0		C	1 1 0 0 1	Rc
fmsubs*x*	1 1 1 0 1 1	D		A	B		C	1 1 1 0 0	Rc
fmadds*x*	1 1 1 0 1 1	D		A	B		C	1 1 1 0 1	Rc
fnmsubs*x*	1 1 1 0 1 1	D		A	B		C	1 1 1 1 0	Rc
fnmadds*x*	1 1 1 0 1 1	D		A	B		C	1 1 1 1 1	Rc
psq_st	1 1 1 1 0 0	S		A	w	i	d		
psq_stu	1 1 1 1 0 1	S		A	w	i	d		
fcmpu	1 1 1 1 1 1	crfD	0 0	A	B		0 0 0 0 0 0 0 0 0 0		0
frsp*x*	1 1 1 1 1 1	D		0 0 0 0 0	B		0 0 0 0 0 0 1 1 0 0		Rc
fctiw*x*	1 1 1 1 1 1	D		0 0 0 0 0	B		0 0 0 0 0 0 1 1 1 0		
fctiwz*x*	1 1 1 1 1 1	D		0 0 0 0 0	B		0 0 0 0 0 0 1 1 1 1		Rc
fdiv*x*	1 1 1 1 1 1	D		A	B		0 0 0 0 0	1 0 0 1 0	Rc
fsub*x*	1 1 1 1 1 1	D		A	B		0 0 0 0 0	1 0 1 0 0	Rc
fadd*x*	1 1 1 1 1 1	D		A	B		0 0 0 0 0	1 0 1 0 1	Rc
fsel*x*	1 1 1 1 1 1	D		A	B		C	1 0 1 1 1	Rc
fmul*x*	1 1 1 1 1 1	D		A	0 0 0 0 0		C	1 1 0 0 1	Rc
frsqrte*x*	1 1 1 1 1 1	D		0 0 0 0 0	B		0 0 0 0 0	1 1 0 1 0	Rc
fmsub*x*	1 1 1 1 1 1	D		A	B		C	1 1 1 0 0	Rc
fmadd*x*	1 1 1 1 1 1	D		A	B		C	1 1 1 0 1	Rc
fnmsub*x*	1 1 1 1 1 1	D		A	B		C	1 1 1 1 0	Rc
fnmadd*x*	1 1 1 1 1 1	D		A	B		C	1 1 1 1 1	Rc
fcmpo	1 1 1 1 1 1	crfD	0 0	A	B		0 0 0 0 1 0 0 0 0 0		0
mtfsb1*x*	1 1 1 1 1 1	crbD		0 0 0 0 0	0 0 0 0 0		0 0 0 0 1 0 0 1 1 0		Rc
fneg*x*	1 1 1 1 1 1	D		0 0 0 0 0	B		0 0 0 0 1 0 1 0 0 0		Rc
mcrfs	1 1 1 1 1 1	crfD	0 0	crfS 0 0	0 0 0 0 0		0 0 0 1 0 0 0 0 0 0		0
mtfsb0*x*	1 1 1 1 1 1	crbD		0 0 0 0 0	0 0 0 0 0		0 0 0 1 0 0 0 1 1 0		Rc
fmr*x*	1 1 1 1 1 1	D		0 0 0 0 0	B		0 0 0 1 0 0 1 0 0 0		Rc
mtfsfi*x*	1 1 1 1 1 1	crfD	0 0	0 0 0 0 0	IMM	0	0 0 1 0 0 0 0 1 1 0		Rc

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
fnabs*x*	1 1 1 1 1 1	D	0 0 0 0 0	B	0 0 1 0 0 0 1 0 0 0	Rc
fabs*x*	1 1 1 1 1 1	D	0 0 0 0 0	B	0 1 0 0 0 0 1 0 0 0	Rc
mffs*x*	1 1 1 1 1 1	D	0 0 0 0 0	0 0 0 0 0	1 0 0 1 0 0 0 1 1 1	Rc
mtfsf*x*	1 1 1 1 1 1	0 FM	0	B	1 0 1 1 0 0 0 1 1 1	Rc

A.2 Instructions Grouped by Functional Categories

Table A-2 through Table A-32 list the Gekko instructions grouped by function.

Key: Reserved bits

Table A-2 Integer Arithmetic Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30	31
add*x*	31	D	A	B	OE	266	Rc
addc*x*	31	D	A	B	OE	10	Rc
adde*x*	31	D	A	B	OE	138	Rc
addi	14	D	A	SIMM			
addic	12	D	A	SIMM			
addic.	13	D	A	SIMM			
addis	15	D	A	SIMM			
addme*x*	31	D	A	0 0 0 0 0	OE	234	Rc
addze*x*	31	D	A	0 0 0 0 0	OE	202	Rc
divw*x*	31	D	A	B	OE	491	Rc
divwu*x*	31	D	A	B	OE	459	Rc
mulhw*x*	31	D	A	B	0	75	Rc
mulhwu*x*	31	D	A	B	0	11	Rc
mulli	07	D	A	SIMM			
mullw*x*	31	D	A	B	OE	235	Rc
neg*x*	31	D	A	0 0 0 0 0	OE	104	Rc
subf*x*	31	D	A	B	OE	40	Rc
subfc*x*	31	D	A	B	OE	8	Rc
subfe*x*	31	D	A	B	OE	136	Rc
subfic*x*	08	D	A	SIMM			
subfme*x*	31	D	A	0 0 0 0 0	OE	232	Rc
subfze*x*	31	D	A	0 0 0 0 0	OE	200	Rc

Table A-3 Integer Compare Instructions

Name	0 5	6 7 8	9	10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
cmp	31	crfD	0	L	A	B	0	0
cmpi	11	crfD	0	L	A	SIMM		

cmpl	31	crfD	0	L	A	B	32	0
cmpli	10	crfD	0	L	A	UIMM		

Table A-4 Integer Logical Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
and*x*	31	S						A					B					28									Rc
andc*x*	31	S						A					B					60									Rc
andi.	28	S						A					UIMM														
andis.	29	S						A					UIMM														
cntlzw*x*	31	S						A					0 0 0 0 0					26									Rc
eqv*x*	31	S						A					B					284									Rc
extsb*x*	31	S						A					0 0 0 0 0					954									Rc
extsh*x*	31	S						A					0 0 0 0 0					922									Rc
nand*x*	31	S						A					B					476									Rc
nor*x*	31	S						A					B					124									Rc
or*x*	31	S						A					B					444									Rc
orc*x*	31	S						A					B					412									Rc
ori	24	S						A					UIMM														
oris	25	S						A					UIMM														
xor*x*	31	S						A					B					316									Rc
xori	26	S						A					UIMM														
xoris	27	S						A					UIMM														

Table A-5 Integer Rotate Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
rlwimi*x*	20	S						A					SH					MB					ME				Rc
rlwinm*x*	21	S						A					SH					MB					ME				Rc
rlwnm*x*	23	S						A					SH					MB					ME				Rc

Table A-6 Integer Shift Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
slw*x*	31	S						A					B					24									Rc
sraw*x*	31	S						A					B					792									Rc

srawi*x*	31	S	A	SH	824	Rc
srw*x*	31	S	A	B	536	Rc

Table A-7 Floating-Point Arithmetic Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
fadd*x*	63	D	A	B	0 0 0 0 0	21	Rc
fadds*x*	59	D	A	B	0 0 0 0 0	21	Rc
fdiv*x*	63	D	A	B	0 0 0 0 0	18	Rc
fdivs*x*	59	D	A	B	0 0 0 0 0	18	Rc
fmul*x*	63	D	A	0 0 0 0 0	C	25	Rc
fmuls*x*	59	D	A	0 0 0 0 0	C	25	Rc
fres*x*	59	D	0 0 0 0 0	B	0 0 0 0 0	24	Rc
frsqrte*x*	63	D	0 0 0 0 0	B	0 0 0 0 0	26	Rc
fsub*x*	63	D	A	B	0 0 0 0 0	20	Rc
fsubs*x*	59	D	A	B	0 0 0 0 0	20	Rc
fsel*x*	63	D	A	B	C	23	Rc

Table A-8 Floating-Point Multiply-Add Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
fmadd*x*	63	D	A	B	C	29	Rc
fmadds*x*	59	D	A	B	C	29	Rc
fmsub*x*	63	D	A	B	C	28	Rc
fmsubs*x*	59	D	A	B	C	28	Rc
fnmadd*x*	63	D	A	B	C	31	Rc
fnmadds*x*	59	D	A	B	C	31	Rc
fnmsub*x*	63	D	A	B	C	30	Rc
fnmsubs*x*	59	D	A	B	C	30	Rc

Table A-9 Floating-Point Rounding and Conversion Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
fctiw*x*	63	D	0 0 0 0 0	B	14		Rc
fctiwz*x*	63	D	0 0 0 0 0	B	15		Rc
frsp*x*	63	D	0 0 0 0 0	B	12		Rc

Table A-10 Floating-Point Compare Instructions

Name	0 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
fcmpo	63	crfD	0 0	A	B	32		0
fcmpu	63	crfD	0 0	A	B	0		0

Table A-11 Floating-Point Status and Control Register Instructions

Name	0 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
mcrfs	63	crfD	0 0	crfS	0 0	0 0 0 0 0	64	0
mffs*x*	63	D		0 0 0 0 0	0 0 0 0 0	583		Rc
mtfsb0*x*	63	crbD		0 0 0 0 0	0 0 0 0 0	70		Rc
mtfsb1*x*	63	crbD		0 0 0 0 0	0 0 0 0 0	38		Rc
mtfsf*x*	63	0	FM		0	B	711	Rc
mtfsfi*x*	63	crfD	0 0	0 0 0 0 0	IMM	0	134	Rc

Table A-12 Integer Load Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
lbz	34	D	A	d			
lbzu	35	D	A	d			

lbzux	31	D	A	B	119	0
lbzx	31	D	A	B	87	0
lha	42	D	A	d		
lhau	43	D	A	d		
lhaux	31	D	A	B	375	0
lhax	31	D	A	B	343	0
lhz	40	D	A	d		
lhzu	41	D	A	d		
lhzux	31	D	A	B	311	0
lhzx	31	D	A	B	279	0
lwz	32	D	A	d		
lwzu	33	D	A	d		
lwzux	31	D	A	B	55	0
lwzx	31	D	A	B	23	0

Table A-13 Integer Store Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
stb	38	S	A	d		
stbu	39	S	A	d		
stbux	31	S	A	B	247	0
stbx	31	S	A	B	215	0
sth	44	S	A	d		
sthu	45	S	A	d		
sthux	31	S	A	B	439	0
sthx	31	S	A	B	407	0
stw	36	S	A	d		
stwu	37	S	A	d		
stwux	31	S	A	B	183	0
stwx	31	S	A	B	151	0

Table A-14 Integer Load and Store with Byte Reverse Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
lhbrx	31	D	A	B	790	0
lwbrx	31	D	A	B	534	0
sthbrx	31	S	A	B	918	0
stwbrx	31	S	A	B	662	0

Table A-15 Integer Load and Store Multiple Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
lmw	46	D	A	d
stmw	47	S	A	d

Table A-16 Integer Load and Store String Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
lswi	31	D	A	NB	597	0
lswx	31	D	A	B	533	0
stswi	31	S	A	NB	725	0
stswx	31	S	A	B	661	0

Table A-17 Memory Synchronization Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
eieio	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	854	0
isync	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	150	0
lwarx	31	D	A	B	20	0
stwcx.	31	S	A	B	150	1
sync	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	598	0

Table A-18 Floating-Point Load Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
lfd	50	D	A	d		
lfdu	51	D	A	d		
lfdux	31	D	A	B	631	0
lfdx	31	D	A	B	599	0
lfs	48	D	A	d		
lfsu	49	D	A	d		
lfsux	31	D	A	B	567	0
lfsx	31	D	A	B	535	0

Table A-19 Floating-Point Store Instructions

Name	0	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
stfd	54		S	A	d		
stfdu	55		S	A	d		
stfdux	31		S	A	B	759	0
stfdx	31		S	A	B	727	0
stfiwx	31		S	A	B	983	0
stfs	52		S	A	d		
stfsu	53		S	A	d		
stfsux	31		S	A	B	695	0
stfsx	31		S	A	B	663	0

Table A-20 Floating-Point Move Instructions

Name	0	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
fabs*x*	63		D	0 0 0 0 0	B	264	Rc
fmr*x*	63		D	0 0 0 0 0	B	72	Rc
fnabs*x*	63		D	0 0 0 0 0	B	136	Rc
fneg*x*	63		D	0 0 0 0 0	B	40	Rc

Table A-21 Branch Instructions

Name	0	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
b*x*	18		LI				AA LK
bc*x*	16		BO	BI	BD		AA LK
bcctr*x*	19		BO	BI	0 0 0 0 0	528	LK
bclr*x*	19		BO	BI	0 0 0 0 0	16	LK

Table A-22 Condition Register Logical Instructions

Name	0–5	6–8	9–10	11–15	16–20	21–30	31
crand	19	crbD		crbA	crbB	257	0
crandc	19	crbD		crbA	crbB	129	0
creqv	19	crbD		crbA	crbB	289	0
crnand	19	crbD		crbA	crbB	225	0
crnor	19	crbD		crbA	crbB	33	0
cror	19	crbD		crbA	crbB	449	0
crorc	19	crbD		crbA	crbB	417	0
crxor	19	crbD		crbA	crbB	193	0
mcrf	19	crfD	0 0	crfS	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0	0

Table A-23 System Linkage Instructions

Name	0–5	6–8	9–10	11–15	16–20	21–29	30	31
rfi [a]	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	50		0
sc	17	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		1	0

Notes:

a. Supervisor-level instruction

Table A-24 Trap Instructions

Name	0–5	6–8	9–10	11–15	16–20	21–30	31
tw	31	TO		A	B	4	0
twi	03	TO		A	SIMM		

Table A-25 Processor Control Instructions

Name	0 – 5	6 – 7	8 – 9	10	11 – 12	13 – 14	15 – 16	17 – 18	19 – 20	21 – 30	31
mcrxr	31	crfS		0 0	0 0 0 0 0		0 0 0 0 0			512	0
mfcr	31	D			0 0 0 0 0		0 0 0 0 0			19	0
mfmsr [a]	31	D			0 0 0 0 0		0 0 0 0 0			83	0
mfspr [b]	31	D			spr					339	0
mftb	31	D			tpr					371	0
mtcrf	31	S		0	CRM				0	144	0
mtmsr [1]	31	S			0 0 0 0 0		0 0 0 0 0			146	0
mtspr [2]	31	D			spr					467	0

Notes:

a. Supervisor-level instruction

b. Supervisor- and user-level instruction

Table A-26 Cache Management Instructions

Name	0 – 5	6 – 7	8 – 9	10	11 – 12	13 – 14	15 – 16	17 – 18	19 – 20	21 – 30	31
dcbf	31	0 0 0 0 0			A		B			86	0
dcbi [a]	31	0 0 0 0 0			A		B			470	0
dcbst	31	0 0 0 0 0			A		B			54	0
dcbt	31	0 0 0 0 0			A		B			278	0
dcbtst	31	0 0 0 0 0			A		B			246	0
dcbz	31	0 0 0 0 0			A		B			1014	0
icbi	31	0 0 0 0 0			A		B			982	0

Notes:

a. Supervisor-level instruction

Table A-27 Segment Register Manipulation Instructions.

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mfsr [a]	31		D					0		SR			0 0 0 0 0					595										0
mfsrin [1]	31		D					0 0 0 0 0					B					659										0
mtsr [1]	31		S					0		SR			0 0 0 0 0					210										0
mtsrin [1]	31		S					0 0 0 0 0					B					242										0

Notes:

a. Supervisor-level instruction

Table A-28 Lookaside Buffer Management Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
tlbie [1]	31		0 0 0 0 0					0 0 0 0 0					B					306										0
tlbsync[1]	31		0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					566										0

Notes:

Table A-29 External Control Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
eciwx	31		D					A					B					310										0
ecowx	31		S					A					B					438										0

Table A-30 Paired-Single Load and Store Instructions

Name	05	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25	26 27 28 29 30	31
psq_lx	4	D	A	B	w	i	6	0
psq_stx	4	S	A	B	w	i	7	0
psq_lux	4	D	A	B	w	i	38	0
psq_stux	4	S	A	B	w	i	39	0
psq_l	56	D	A	w	i	d		
psq_lu	57	D	A	w	i	d		
psq_st	60	S	A	w	i	d		
psq_stu	61	S	A	w	i	d		

Table A-31 Paired-Single Floating Point Arithmetic Instructions

Name	05	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
ps_div	4	D	A	B	0 0 0 0 0	18	Rc
ps_sub	4	D	A	B	0 0 0 0 0	20	Rc
ps_add	4	D	A	B	0 0 0 0 0	21	Rc
ps_sel	4	D	A	B	C	23	Rc
ps_res	4	D	00000	B	00000	24	Rc
ps_mul	4	D	A	00000	C	25	Rc
ps_rsqrte	4	D	00000	B	00000	26	Rc
ps_msub	4	D	A	B	C	28	Rc
ps_madd	4	D	A	B	C	29	Rc
ps_nmsub	4	D	A	B	C	30	Rc
ps_nmadd	4	D	A	B	C	31	Rc
ps_neg	4	D	00000	B	40		Rc
ps_mr	4	D	00000	B	72		Rc
ps_nabs	4	D	00000	B	136		Rc
ps_abs	4	D	00000	B	264		Rc

Table A-32 Miscellaneous Paired-Single Instructions

Name	05	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
ps_sum0	4	D	A	B	C	10	Rc
ps_sum1	4	D	A	B	C	11	Rc
ps_muls0	4	D	A	0 0 0 0 0	C	12	Rc
ps_muls1	4	D	A	0 0 0 0 0	C	13	Rc
ps_madds0	4	D	A	B	C	14	Rc
ps_madds1	4	D	A	B	C	15	Rc
ps_cmpu0	4	crfD 00	A	B	0		0
ps_cmpo0	4	crfD 00	A	B	32		0
ps_cmpu1	4	crfD 00	A	B	64		0
ps_cmpo1	4	crfD 00	A	B	96		0
ps_merge00	4	D	A	B	528		Rc
ps_merge01	4	D	A	B	560		Rc
ps_merge10	4	D	A	B	592		Rc
ps_merge11	4	D	A	B	624		Rc
dcbz_l	4	00000	A	B	1014		0

Index

C

D

E

F

G

H

I

INDEX (Continued)

M

N

O

P

Q

R

S

T

U

V

W

X