SECTION 4
ADDRESSING MODES AND INSTRUCTION SET SUMMARY

This section describes instructions and address modes supported by the RCPU. These instructions are divided into the following categories:

- Integer instructions — These include computational and logical instructions.
- Floating-point instructions — These include floating-point computational instructions, as well as instructions that affect the floating-point status and control register.
- Load/store instructions — These include integer and floating-point load and store instructions.
- Flow control instruction — These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow.
- Processor control instruction — These instructions are used to read from and write to the condition register (CR), machine state register (MSR), and special-purpose registers (SPRs), and to read from the time base register (TBU or TBL).
- Memory synchronization instructions — These instructions are used for synchronizing memory.
- Memory control instructions — These instructions provide control of the I-cache.

Notice that this grouping of instructions does not necessarily indicate the execution unit that processes a particular instruction or group of instructions. This information is provided in **SECTION 9 INSTRUCTION SET**.

Integer instructions operate on byte, half-word, and word operands. Floating-point instructions operate on single-precision and double-precision 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 fetches and stores between memory and a set of 32 general-purpose registers (GPRs). It also provides for word and double-word operand fetches and stores between memory and a set of 32 floating-point registers (FPRs).

Arithmetic and logical 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.

4.1 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 sequential instruction.

4.1.1 Memory Operands

Integer instructions operate on byte, half-word, and word operands. Floating-point instructions operate on single-precision and double-precision floating-point operands. The address of a memory operand is the address 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 **3.2 Byte Ordering** for more information.

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 of memory operands, see **SECTION 3 OPERAND CONVENTIONS**.

4.1.2 Addressing Modes and Effective Address Calculation

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

The effective address is the 32-bit address 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 storage operand is considered to wrap around from the maximum effective address to 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 three categories of effective address generation:

- Register indirect with immediate index mode. The d operand is added to the contents of the GPR specified by the **r**A operand to generate the effective address.
- Register indirect with index mode. The contents of the GPR specified by **r**B operand are added to the contents of the GPR specified by the **r**A operand to generate the effective address.
- Register indirect mode. The contents of the GPR specified by the **r**A operand are used as the effective address.

Branch instructions have three categories of effective address generation:

- Immediate addressing. The BD or LI operands are sign extended with the two low-order bits cleared to zero to generate the branch effective address.
- Link register indirect. The contents of the link register with the two low-order bits cleared to zero are used as the branch effective address.
- Counter register indirect. The contents of the counter register with the two low-order bits cleared to zero are used as the branch effective address.

Branch instructions can optionally load the link register with the next sequential instruction address (current instruction address + 4).

4.2 Classes of Instructions

PowerPC instructions belong to one of 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, an instruction that is specific to 64-bit implementations is considered defined for 64-bit implementations but illegal for 32-bit implementations such as the RCPU.

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 a reserved instruction, the instruction is illegal.

In future versions of the PowerPC architecture, instruction codings that are now illegal may become defined (by being added to the architecture) or reserved (by being assigned to one of the special purposes). Likewise, reserved instructions may become defined.

4.2.1 Definition of Boundedly Undefined

The results of executing a given instruction are said to be boundedly undefined if they could have been achieved by executing an arbitrary sequence of instructions, starting in the state the machine was in before executing the given instruction. Boundedly undefined results for a given instruction may vary between implementations and between execution attempts on the same implementation.

4.2.2 Defined Instruction Class

Defined instructions include all the instructions defined in the PowerPC UISA, VEA, and OEA. Defined instructions can be required or optional. The RCPU supports the following defined instructions:

- All 32-bit PowerPC UISA required instructions
- The following PowerPC VEA instructions: **eieio**, **icbi**, **isync**, and **mftb**
- The following PowerPC OEA instructions: **mfmsr**, **mfspr**, **mtmsr**, **mtspr**, **rfi**, and **sc**.
- The following optional instruction: **stfiwx**

A defined instruction may have an instruction form that is invalid if one or more operands, excluding opcodes, are coded incorrectly in a manner that can be deduced by examining only the instruction encoding (primary and extended opcodes). For example, an invalid form results when a reserved bit (shown as “0” in the instruction descriptions in **SECTION 9 INSTRUCTION SET**) is set to one.

Attempting to execute an invalid form of a defined instruction either invokes the software emulation instruction error handler or yields boundedly undefined results. Where not otherwise noted in the individual instruction descriptions in **SECTION 9 INSTRUCTION SET** for individual instruction descriptions, attempting to execute

an instruction in which a reserved bit is set to one yields the same result as executing the instruction with the reserved bit cleared to zero.

Attempting to execute a defined PowerPC instruction, including an optional instruction, that is not implemented in hardware causes the RCPU to take the implementation dependent software emulation exception.

NOTE

Other PowerPC implementations invoke the program exception handler in this case. Refer to **6.11.11 Software Emulation Exception (0x01000)** for additional information.

4.2.3 Illegal Instruction Class

Illegal instructions can be grouped into the following categories:

- Instructions that are not implemented in the PowerPC architecture. These opcodes are available for future extensions of the PowerPC architecture; that is, future versions of the PowerPC architecture may define any of these instructions to perform new functions.
- Instructions that are implemented in the PowerPC architecture but are not implemented in a specific PowerPC implementation. For example, instructions that can be executed on 64-bit PowerPC processors are considered illegal for 32-bit processors.
- All unused extended opcodes are illegal.
- An instruction consisting entirely of zeros is guaranteed to be an illegal instruction.

An attempt to execute an illegal instruction invokes the software emulation error handler. Notice that in other PowerPC implementations, the program exception handler may be invoked in this case.

4.2.4 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 causes the RCPU to take the implementation dependent software emulation exception.

NOTE

Other PowerPC implementations invoke the program exception handler in this case. Refer to **6.11.11 Software Emulation Exception (0x01000)** for additional information.

4.3 Integer Instructions

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

- Integer arithmetic instructions
- Integer compare instructions

- Integer rotate and shift instructions
- Integer logical 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 fields.

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

The integer instructions that update the condition register (i.e., those with a mnemonic ending in a period) set condition register field CR0 (bits [0:3]) to characterize the result of the operation. These instructions include those with the Rc bit equal to one and the **addic.**, **andi.**, and **andis.** integer logical and arithmetic instructions. The condition register field CR0 is set as if the result were compared algebraically to zero.

The following integer arithmetic instructions always set XER[CA] to reflect the carry out of bit 0: **addic**, **addic.**, **subfic**, **addc**, **subfc**, **adde**, **subfe**, **addme**, **subfme**, **addze**, and **subfze**. Integer arithmetic instructions with the overflow enable (OE) bit set cause XER[SO] and XER[OV] to be set to reflect overflow of the 32-bit result.

Unless otherwise noted, when condition register field CR0 and the XER are affected, they reflect the value placed in the target register.

The RCPU performs best for aligned load and store operations. See**6.11.4 Alignment Exception (0x00600)** for scenarios that cause an alignment exception.

4.3.1 Integer Arithmetic Instructions

Table 4-1 lists the integer arithmetic instructions.

Table 4-1 Integer Arithmetic Instructions

Name	Mnemonic	Operand Syntax	Operation
Add Immediate	**addi**	**r**D,**r**A,SIMM	The sum (**r**A|0) + SIMM is placed into register **r**D.
Add Immediate Shifted	**addis**	**r**D,**r**A,SIMM	The sum (**r**A|0) + (SIMM || 0x0000) is placed into register **r**D.
Add	**add** **add.** **addo** **addo.**	**r**D,**r**A,**r**B	The sum (**r**A) + (**r**B) is placed into register **r**D. **add** **Add** **add.** **Add** with CR Update. The dot suffix enables the update of the condition register. **addo** **Add** with Overflow Enabled. The o suffix enables the overflow bit (OV) in the XER. **addo.** **Add** with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.
Subtract from	**subf** **subf.** **subfo** **subfo.**	**r**D,**r**A,**r**B	The sum ¬ (**r**A) + (**r**B) +1 is placed into **r**D. **subf** Subtract from **subf.** Subtract from with CR Update. The dot suffix enables the update of the condition register. **subfo** Subtract from with Overflow Enabled. The o suffix enables the overflow. The o suffix enables the overflow bit (OV) in the XER. **subfo.** Subtract from with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.
Add Immediate Carrying	**addic**	**r**D,**r**A,SIMM	The sum (**r**A) + SIMM is placed into register **r**D.
Add Immediate Carrying and Record	**addic.**	**r**D,**r**A,SIMM	The sum (**r**A) + SIMM is placed into **r**D. The condition register is updated.
Subtract from Immediate Carrying	**subfic**	**r**D,**r**A,SIMM	The sum ¬ (**r**A) + SIMM + 1 is placed into register **r**D.
Add Carrying	**addc** **addc.** **addco** **addco.**	**r**D,**r**A,**r**B	The sum (**r**A) + (**r**B) is placed into register **r**D. **addc** Add Carrying **addc.** Add Carrying with CR Update. The dot suffix enables the update of the condition register. **addco** Add Carrying with Overflow Enabled. The o suffix enables the overflow bit (OV) in the XER. **addco.** Add Carrying with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.

Table 4-1 Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Subtract from Carrying	**subfc** **subfc.** **subfco** **subfco.**	**r**D,**r**A,**r**B	The sum ¬ (**r**A) + (**r**B) + 1 is placed into register **r**D. **subfc** Subtract from Carrying **subfc.** Subtract from Carrying with CR Update. The dot suffix enables the update of the condition register. **subfco** Subtract from Carrying with Overflow. The o suffix enables the overflow bit (OV) in the XER. **subfco.** Subtract from Carrying with Overflow and CR Updat e. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.
Add Extended	**adde** **adde.** **addeo** **addeo.**	**r**D,**r**A,**r**B	The sum (**r**A) + (**r**B) + XER(CA) is placed into register **r**D. **adde** Add Extended **adde.** Add Extended with CR Update. The dot suffix enables the update of the condition register. **addeo** Add Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER. **addeo.** Add Extended with Overflow and CR Update .The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.
Subtract from Extended	**subfe** **subfe.** **subfeo** **subfeo.**	**r**D,**r**A,**r**B	The sum ¬ (**r**A) + (**r**B) + XER(CA) is placed into register **r**D. **subfe** Subtract from Extended **subfe.** Subtract from Extended with CR Update. The dot suffix enables the update of the condition register. **subfeo** Subtract from Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER. **subfeo.** Subtract from Extended with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow (OV) bit in the XER.
Add to Minus One Extended	**addme** **addme.** **addmeo** **addmeo.**	**r**D,**r**A	The sum (**r**A) + XER(CA) + 0xFFFF FFFF is placed into register **r**D. **addme** Add to Minus One Extended **addme.** Add to Minus One Extended with CR Update .The dot suffix enables the update of the condition register. **addmeo** Add to Minus One Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER. **addmeo.** Add to Minus One Extended with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow (OV) bit in the XER.
Subtract from Minus One Extended	**subfme** **subfme.** **subfmeo** **subfmeo.**	**r**D,**r**A	The sum ¬ (**r**A) + XER(CA) + 0xFFFF FFFF is placed into register **r**D. **subfme** Subtract from Minus One Extended **subfme.** Subtract from Minus One Extended with CR Update. The dot suffix enables the update of the condition register. **subfmeo** Subtract from Minus One Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER. **subfmeo.** Subtract from Minus One Extended with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.

Table 4-1 Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Add to Zero Extended	**addze** **addze.** **addzeo** **addzeo.**	**r**D,**r**A	The sum (**r**A) + XER(CA) is placed into register **r**D. **addze** Add to Zero Extended **addze.** Add to Zero Extended with CR Update. The dot suffix enables the update of the condition register. **addzeo** Add to Zero Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER. **addzeo.** Add to Zero Extended with Overflow and CR Update . The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.
Subtract from Zero Extended	**subfze** **subfze.** **subfzeo** **subfzeo.**	**r**D,**r**A	The sum ¬ (**r**A) + XER(CA) is placed into register **r**D. **subfze** Subtract from Zero Extended **subfze.** Subtract from Zero Extended with CR Updat e.The dot suffix enables the update of the condition register. **subfzeo** Subtract from Zero Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER. **subfzeo.** Subtract from Zero Extended with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.
Negate	**neg** **neg.** **nego** **nego.**	**r**D,**r**A	The sum ¬ (**r**A) + 1 is placed into register **r**D. **neg** Negate **neg.** Negate with CR Update. The dot suffix enables the update of the condition register. **nego** Negate with Overflow. The o suffix enables the overflow bit (OV) in the XER. **nego.** Negate with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.
Multiply Low Immediate	**mulli**	**r**D,**r**A,SIMM	The low-order 32 bits of the 48-bit product (**r**A) * SIMM are placed into register **r**D. The low-order 32 bits of the product are the correct 32-bit product. The low-order bits are independent of whether the operands are treated as signed or unsigned integers. However, XER[OV] is set based on the result interpreted as a signed integer. The high-order bits are lost. This instruction can be used with **mulhw***x* to calculate a full 64-bit product.

Table 4-1 Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Multiply Low	**mullw** **mullw.** **mullwo** **mullwo.**	**r**D,**r**A,**r**B	The low-order 32 bits of the 64-bit product (**r**A) * (**r**B) are placed into register **r**D. The low-order 32 bits of the product are the correct 32-bit product. The low-order bits are independent of whether the operands are treated as signed or unsigned integers. However, XER[OV] is set based on the result interpreted as a signed integer. The high-order bits are lost. This instruction can be used with **mulhw***x* to calculate a full 64-bit product. Some implementations may execute faster if **r**B contains the operand having the smaller absolute value. **mullw** Multiply Low **mullw.** Multiply Low with CR Update. The dot suffix enables the update of the condition register. **mullwo** Multiply Low with Overflow. The o suffix enables the overflow bit (OV) in the XER. **mullwo.** Multiply Low with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.
Multiply High Word	**mulhw** **mulhw.**	**r**D,**r**A,**r**B	The contents of **r**A and **r**B are interpreted as 32-bit signed integers. The 64-bit product is formed. The high-order 32 bits of the 64-bit product are placed into **r**D. Both operands and the product are interpreted as signed integers. This instruction may execute faster if **r**B contains the operand having the smaller absolute value. **mulhw** Multiply High Word **mulhw.** Multiply High Word with CR Update. The dot suffix enables the update of the condition register.
Multiply High Word Unsigned	**mulhwu** **mulhwu.**	**r**D,**r**A,**r**B	The contents of **r**A and of **r**B are extracted and interpreted as 32-bit unsigned integers. The 64-bit product is formed. The high-order 32 bits of the 64-bit product are placed into **r**D. Both operands and the product are interpreted as unsigned integers. This instruction may execute faster if **r**B contains the operand having the smaller absolute value. **mulhwu** Multiply High Word Unsigned **mulhwu.** Multiply High Word Unsigned with CR Update. The dot suffix enables the update of the condition register.

Table 4-1 Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Divide Word	**divw** **divw.** **divwo** **divwo.**	**r**D,**r**A,**r**B	The dividend is the signed value of (**r**A). The divisor is the signed value of (**r**B). The 64-bit quotient is formed. The low-order 32 bits of the 64-bit quotient are placed into **r**D. The remainder is not supplied as a result. Both operands are interpreted as signed integers. The quotient is the unique signed integer that satisfies the following: dividend = (quotient times divisor) + r where 0 ð r < |divisor| if the dividend is non-negative, and –|divisor| < r ð 0 if the dividend is negative. If an attempt is made to perform any of the divisions 0x8000 0000 / –1 or <anything> / 0 the contents of register **r**D are undefined, as are the contents of the LT, GT, and EQ bits of the condition register field CR0 if the instruction has condition register updating enabled. In these cases, if instruction overflow is enabled, then XER[OV] is set. The 32-bit signed remainder of dividing (**r**A) by (**r**B) can be computed as follows, except in the case that (**r**A) = -2^{31} and (**r**B) = –1: **divw** **r**D,**r**A,**r**B **r**D = quotient **mull** **r**D,**r**D,**r**B **r**D = quotient*divisor **subf** **r**D,**r**D,**r**A **r**D = remainder **divw** Divide Word **divw.** Divide Word with CR Update. The dot suffix enables the update of the condition register. **divwo** Divide Word with Overflow. The o suffix enables the overflow bit (OV) in the XER. **divwo.** Divide Word with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.

Table 4-1 Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Divide Word Unsigned	**divwu** **divwu.** **divwuo** **divwuo.**	**r**D,**r**A,**r**B	The dividend is the value of (**r**A). The divisor is the value of (**r**B). The 32-bit quotient is placed into **r**D. The remainder is not supplied as a result. Both operands are interpreted as unsigned integers. The quotient is the unique unsigned integer that satisfies the following: dividend = (quotient times divisor) + r where 0 ð r < divisor. If an attempt is made to perform the division <anything> / 0 the contents of register **r**D are undefined, as are the contents of the LT, GT, and EQ bits of the condition register field CR0 if the instruction has the condition register updating enabled. In these cases, if instruction overflow is enabled, then XER[OV] is set. The 32-bit unsigned remainder of dividing (**r**A) by (**r**B) can be computed as follows: **divwu** **r**D,**r**A,**r**B **r**D = quotient **mull** **r**D,**r**D,**r**B **r**D = quotient*divisor **subf** **r**D,**r**D,**r**A **r**D = remainder **divwu** Divide Word Unsigned **divwu.** Divide Word Unsigned with CR Update. The dot suffix enables the update of the condition register. **divwuo** Divide Word Unsigned with Overflow. The o suffix enables the overflow bit (OV) in the XER. **divwuo.** Divide Word Unsigned with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.

See **E.2 Simplified Mnemonics for Subtract Instructions** for information on simplified mnemonics.

4.3.2 Integer Compare Instructions

The integer compare instructions algebraically or logically compare the contents of register **r**A with either the **UIMM** operand, the **SIMM** operand or the contents of register **r**B. Algebraic comparison compares two signed integers. Logical comparison compares two unsigned numbers. **Table 4-2** summarizes the RCPU integer compare instructions.

Table 4-2 Integer Compare Instructions

Name	Mnemonic	Operand Syntax	Operation
Compare Immediate	**cmpi**	crfD,L,rA,SIMM	The contents of register **r**A is compared with the sign-extended value of the SIMM operand, treating the operands as signed integers. The result of the comparison is placed into the CR field specified by operand **crf**D.
Compare	**cmp**	**crf**D,L,**r**A,**r**B	The contents of register **r**A is compared with register **r**B, treating the operands as signed integers. The result of the comparison is placed into the CR field specified by operand **crf**D.
Compare Logical Immediate	**cmpli**	**crf**D,L,**r**A,UIMM	The contents of register **r**A is compared with 0x0000 || UIMM, treating the operands as unsigned integers. The result of the comparison is placed into the CR field specified by operand **crf**D.
Compare Logical	**cmpl**	**crf**D,L,**r**A,**r**B	The contents of register **r**A is compared with register **r**B, treating the operands as unsigned integers. The result of the comparison is placed into the CR field specified by operand **crf**D.

While the PowerPC architecture specifies that the value in the L field specifies whether the operands are treated as 32- or 64-bit values, the RCPU ignores the value in the L field and treats the operands as 32-bit values.

The **crf**D field can be omitted if the result of the comparison is to be placed in CR0. Otherwise the target CR field must be specified in the instruction **crf**D field, using one of the CR field symbols (CR0 to CR7) or an explicit field number. Refer to **Table E-2** for the list of CR field symbols and to **E.3 Simplified Mnemonics for Compare Instructions** for simplified mnemonics.

4.3.3 Integer Logical Instructions

The logical instructions shown in **Table 4-4** perform bit-parallel operations. Logical instructions with Rc = 1 and instructions **andi.** and **andis.** set condition register field CR0 to characterize the result of the logical operation. These fields are set as if the sign-extended low-order 32 bits of the result were algebraically compared to zero. The remaining logical instructions do not modify the condition register. Logical instructions do not change the SO, OV, or CA bits in the XER.

Table 4-3 Integer Logical Instructions

Name	Mnemonic	Operand Syntax	Operation
AND Immediate	**andi.**	**r**A,**r**S,UIMM	The contents of **r**S is ANDed with 0x0000 || UIMM and the result is placed into **r**A.
AND Immediate Shifted	**andis.**	**r**A,**r**S,UIMM	The contents of **r**S is ANDed with UIMM || 0x0000 and the result is placed into **r**A.
OR Immediate	**ori**	**r**A,**r**S,UIMM	The contents of **r**S is ORed with 0x0000 || UIMM and the result is placed into **r**A. The preferred no-op is **ori 0,0,0**
OR Immediate Shifted	**oris**	**r**A,**r**S,UIMM	The contents of **r**S is ORed with UIMM || 0x0000 and the result is placed into **r**A.
XOR Immediate	**xori**	**r**A,**r**S,UIMM	The contents of **r**S is XORed with 0x0000 || UIMM and the result is placed into **r**A.
XOR Immediate Shifted	**xoris**	**r**A,**r**S,UIMM	The contents of **r**S is XORed with UIMM || 0x0000 and the result is placed into **r**A.
AND	**and** **and.**	**r**A,**r**S,**r**B	The contents of **r**S is ANDed with the contents of register **r**B and the result is placed into **r**A. **and** AND **and.** AND with CR Update. The dot suffix enables the update of the condition register.
OR	**or** **or.**	**r**A,**r**S,**r**B	The contents of **r**S is ORed with the contents of **r**B and the result is placed into **r**A. **or** OR **or.** OR with CR Update. The dot suffix enables the update of the condition register.
XOR	**xor** **xor.**	**r**A,**r**S,**r**B	The contents of **r**S is XORed with the contents of **r**B and the result is placed into register **r**A. **xor** XOR **xor.** XOR with CR Update. The dot suffix enables the update of the condition register.
NAND	**nand** **nand.**	**r**A,**r**S,**r**B	The contents of **r**S is ANDed with the contents of **r**B and the one's complement of the result is placed into register **r**A. **nand** NAND **nand.** NAND with CR Update. The dot suffix enables the update of the condition register. NAND with **r**S = **r**B can be used to obtain the one's complement.
NOR	**nor** **nor.**	**r**A,**r**S,**r**B	The contents of **r**S is ORed with the contents of **r**B and the one's complement of the result is placed into register **r**A. **nor** NOR **nor.** NOR with CR Update. The dot suffix enables the update of the condition register. NOR with **r**S = **r**B can be used to obtain the one's complement.

Table 4-3 Integer Logical Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Equivalent	**eqv** **eqv.**	**r**A,**r**S,**r**B	The contents of **r**S is XORed with the contents of **r**B and the complemented result is placed into register **r**A. **eqv** Equivalent **eqv.** Equivalent with CR Update**.** The dot suffix enables the update of the condition register.
AND with Complement	**andc** **andc.**	**r**A,**r**S,**r**B	The contents of **r**S is ANDed with the complement of the contents of **r**B and the result is placed into **r**A. **andc** AND with Complement **andc.** AND with Complement with CR Update. The dot suffix enables the update of the condition register.
OR with Complement	**orc** **orc.**	**r**A,**r**S,**r**B	The contents of **r**S is ORed with the complement of the contents of **r**B and the result is placed into **r**A. **orc** OR with Complement **orc.** OR with Complement with CR Update. The dot suffix enables the update of the condition register.
Extend Sign Byte	**extsb** **extsb.**	**r**A,**r**S	The contents of **r**S[24:31] are placed into **r**A[24:31]. Bit 24 of **r**S is placed into **r**A[0:23]. **extsb** Extend Sign Byte **extsb.** Extend Sign Byte with CR Update. The dot suffix enables the update of the condition register.
Extend Sign Half Word	**extsh** **extsh.**	**r**A,**r**S	The contents of **r**S[16:31] are placed into **r**A[16:31]. Bit 16 of **r**S is placed into **r**A[0:15]. **extsh** Extend Sign Half Word **extsh.** Extend Sign Half Word with CR Update. The dot suffix enables the update of the condition register.
Count Leading Zeros Word	**cntlzw** **cntlzw.**	**r**A,**r**S	A count of the number of consecutive zero bits of **r**S is placed into **r**A. This number ranges from 0 to 32, inclusive. **cntlzw** Count Leading Zeros Word **cntlzw.** Count Leading Zeros Word with CR Update. The dot suffix enables the update of the condition register. When the Count Leading Zeros Word instruction has condition register updating enabled, the LT field is cleared to zero in CR0.

4.3.4 Integer Rotate and Shift Instructions

Rotate and shift instructions provide powerful and general ways to manipulate register contents. **Table 4-4** shows the types of rotate and shift operations provided by the RCPU.

Table 4-4 Rotate and Shift Operations

Operation	Description
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.
Insert	Select a left- or right-justified 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 left-justified field when operating on double-words; such an insertion requires more than one instruction.)
Rotate	Rotate the contents of a register right or left *n* bits without masking.
Shift	Shift the contents of a register right or left *n* bits, clearing vacated bits to zero (logical shift).
Clear	Clear the leftmost or rightmost *n* bits of a register to zero.
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.

The IU performs rotation operations on data from a GPR and returns the result, or a portion of the result, to a GPR. Rotation operations rotate a 32-bit quantity left by a specified number of bit positions. Bits that exit from position 0 enter at position 31. A rotate right operation can be accomplished by specifying a rotation of 32-*n* bits, where *n* is the right rotation amount.

Rotate and shift instructions use a mask generator. The mask is 32 bits long and consists of 1-bits from a start bit, **MB**, through and including a stop bit, **ME**, and 0-bits elsewhere. The values of **MB** and **ME** range from zero to 31. If **MB** > **ME**, the 1-bits wrap around from position 31 to position 0. Thus the mask is formed as follows:

if **MB** ð **ME** then

```
    mask[mstart:mstop] = ones
    mask[all other bits] = zeros
else
    mask[mstart:31] = ones
    mask[0:mstop] = ones
    mask[all other bits] = zeros
```

It is not possible to specify an all-zero mask. The use of the mask is described in the following sections.

If condition register updating is enabled, rotate and shift instructions set condition register field CR0 according to the contents of **r**A at the completion of the instruction. Rotate and shift instructions do not change the values of XER[OV] or XER[SO] bits. Rotate and shift instructions, except algebraic right shifts, do not change the XER[CA] bit.

Simplified mnemonics allow simpler coding of often-used functions such as clear-

ing the leftmost or rightmost bits of a register, left justifying or right justifying an arbitrary field, and simple rotates and shifts. Some of these are shown as examples with the rotate instructions. In addition, **E.4 Simplified Mnemonics for Rotate and Shift Instructions** provides a list of these mnemonics.

4.3.4.1 Integer Rotate Instructions

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

Rotate left instructions allow right-rotation of the contents of a register to be performed by a left-rotation of 32 - *n*, where *n* is the number of bits by which to rotate right.

The integer rotate instructions are summarized in **Table 4-5**.

Table 4-5 Integer Rotate Instructions

Name	Mnemonic	Operand Syntax	Operation
Rotate Left Word Immediate then AND with Mask	**rlwinm** **rlwinm.**	**r**A,**r**S,SH,MB,ME	The contents of register **r**S are rotated left by the number of bits specified by operand SH. A mask is generated having 1-bits from the bit specified by operand MB through the bit specified by operand ME and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register **r**A. **rlwinm** Rotate Left Word Immediate then AND with Mask **rlwinm.** Rotate Left Word Immediate then AND with Mask with CR Update. The dot suffix enables the update of the condition register. Simplified mnemonics: **extlwi r**A,**r**S,*n*,b**rlwinm r**A,**r**S,b,**0,***n*-1 **srwi r**A,**r**S,*n***rlwinm r**A,**r**S,32-*n*,*n*,31 **clrrwi r**A,**r**S,*n* **rlwinm r**A,**r**S,**0,0,**31-*n* Note: The **rlwinm** instruction can be used for extracting, clearing and shifting bit fields using the methods shown below: To extract an *n*-bit field that starts at bit position *b* in register **r**S, right-justified into **r**A (clearing the remaining 32 - *n* bits of **r**A), set SH = *b* + *n*, MB = 32 - *n*, and ME = 31. To extract an *n*-bit field that starts at bit position *b* in **r**S, left-justified into **r**A, set SH = *b*, MB = 0, and ME = *n* - 1. To rotate the contents of a register left (right) by *n* bits, set SH = *n* (32 - *n*), MB = 0, and ME = 31. To shift the contents of a register right by *n* bits, set SH = 32 - *n*, MB = *n*, and ME = 31. To clear the high-order *b* bits of a register and then shift the result left by *n* bits, set SH = *n*, MB = *b* - *n* and ME = 31 - *n*. To clear the low-order *n* bits of a register, set SH = 0, MB = 0, and ME = 31 - *n*.

Table 4-5 Integer Rotate Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Rotate Left Word then AND with Mask	**rlwnm** **rlwnm.**	**r**A,**r**S,**r**B,MB,ME	The contents of **r**S are rotated left by the number of bits specified by **r**B[27:31]. A mask is generated having 1-bits from the bit specified by operand MB through the bit specified by operand ME and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A. **rlwinm** Rotate Left Word then AND with Mask **rlwinm.** Rotate Left Word then AND with Mask with CR Update. The dot suffix enables the update of the condition register. Simplified mnemonics: **rotlw r**A,**r**S,**r**B**rlwnm r**A,**r**S,**r**B**,0,31** Note: The **rlwinm** instruction can be used to extract and rotate bit fields using the methods shown below: To extract an *n*-bit field that starts at the variable bit position *b* in the register specified by operand **r**S, right-justified into **r**A (clearing the remaining 32-*n* bits of **r**A), set **r**B[27:31] = *b* + *n*, MB = 32 - *n*, and ME = 31. To extract an *n*-bit field that starts at variable bit position *b* in the register specified by operand **r**S, left-justified into **r**A (clearing the remaining 32 - *n* bits of **r**A), set **r**B[27:31] = *b*, MB = 0, and ME = *n* - 1. To rotate the contents of the low-order 32 bits of a register left (right) by variable *n* bits, set **r**B[27:31] = *n* (32 - n), MB = 0, and ME = 31.
Rotate Left Word Immediate then Mask Insert	**rlwimi** **rlwimi.**	**r**A,**r**S,SH,MB,ME	The contents of **r**S are rotated left by the number of bits specified by operand SH. A mask is generated having 1-bits from the bit specified by MB through the bit specified by ME and 0-bits elsewhere. The rotated data is inserted into **r**A under control of the generated mask. **rlwimi** Rotate Left Word Immediate then Mask **rlwimi.** Rotate Left Word Immediate then Mask Insert with CR Update. The dot suffix enables the update of the condition register. Simplified mnemonic: **inslw r**A,**r**S,*n*,*b***rlwim r**A,**r**S,**32-***b*,*b*,*b*+*n*-**1** Note: The opcode **rlwimi** can be used to insert a bit field into the contents of register specified by operand **r**A using the methods shown below: To insert an *n*-bit field that is left-justified in **r**S into **r**A starting at bit position *b*, set SH = 32 - *b*, MB = *b*, and ME = (*b* + *n*) - 1. To insert an *n*-bit field that is right-justified in **r**S into **r**A starting at bit position *b*, set SH =3 2 - (*b* + *n*), MB = *b*, and ME = (*b* + *n*) - 1. Simplified mnemonics are provided for both of these methods.

4.3.4.2 Integer Shift Instructions

The instructions in this section 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 are provided to make coding of

such shifts simpler and easier to understand.

Any shift right algebraic instruction, followed by **addze**, can be used to divide quickly by 2^n.

Multiple-precision shifts can be programmed as shown in **APPENDIX B MULTIPLE-PRECISION SHIFTS**.

The integer shift instructions are summarized in **Table 4-6**.

Table 4-6 Integer Shift Instructions

Name	Mnemonic	Operand Syntax	Operation
Shift Left Word	**slw** **slw.**	**r**A,**r**S,**r**B	The contents of **r**S are shifted left the number of bits specified by **r**B[26:31]. 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. If **r**B[26] = 1, then **r**A is filled with zeros. **slw** Shift Left Word **slw.** Shift Left Word with CR Update. The dot suffix enables the update of the condition register.
Shift Right Word	**srw** **srw.**	**r**A,**r**S,**r**B	The contents of **r**S are shifted right the number of bits specified by **r**B[26:31]. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into **r**A. If **r**B[26]=1, then **r**A is filled with zeros. **srw** Shift Right Word **srw.** Shift Right Word with CR Update. The dot suffix enables the update of the condition register.
Shift Right Algebraic Word Immediate	**srawi** **srawi.**	**r**A,**r**S,SH	The contents of **r**S are shifted right the number of bits specified by operand SH. Bits shifted out of position 31 are lost. The 32-bit result is sign extended and 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. An operand SH of zero causes **r**A to be loaded with the contents of **r**S and XER[CA] to be cleared to zero. **srawi** Shift Right Algebraic Word Immediate **srawi.** Shift Right Algebraic Word Immediate with CR Update. The dot suffix enables the update of the condition register.
Shift Right Algebraic Word	**sraw** **sraw.**	**r**A,**r**S,**r**B	The contents of **r**S are shifted right the number of bits specified by **r**B[26:31]. The 32-bit result is placed into **r**A. XER[CA] is set to one if **r**S contains a negative number and any 1-bits are shifted out of position 31; otherwise XER[CA] is cleared to zero. An operand (**r**B) of zero causes **r**A to be loaded with the contents of **r**S, and XER[CA] to be cleared to zero. If **r**B[26] = 1, then **r**A is filled with 32 sign bits (bit 0) from **r**S. If **r**B[26] = 0, then **r**A is filled from the left with sign bits. Condition register field CR0 is set based on the value written into**r**A. **sraw** Shift Right Algebraic Word **sraw.** Shift Right Algebraic Word with CR Update. The dot suffix enables the update of the condition register.

4.4 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 loads and stores are discussed in **4.5 Load and Store Instructions**.

4.4.1 Floating-Point Arithmetic Instructions

The floating-point arithmetic instructions are summarized in **Table 4-7**.

Table 4-7 Floating-Point Arithmetic Instructions

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Add	**fadd** **fadd.**	**fr**D,**fr**A,**fr**B	The floating-point operand in register **fr**A is added to the floating-point operand in register **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 target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **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 algebraically to form an intermediate sum. 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. **fadd** Floating-Point Add **fadd.** Floating-Point Add with CR Update. The dot suffix enables the update of the condition register.

Table 4-7 Floating-Point Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Add Single-Precision	**fadds** **fadds.**	**fr**D,**fr**A,**fr**B	The floating-point operand in register **fr**A is added to the floating-point operand in register **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 target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **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 algebraically to form an intermediate sum. 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. **fadds** Floating-Point Single-Precision **fadds.** Floating-Point Single-Precision with CR Update. The dot suffix enables the update of the condition register.
Floating-Point Subtract	**fsub** **fsub.**	**fr**D,**fr**A,**fr**B	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 the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **fr**D. The execution of the Floating-Point Subtract instruction is identical to that of Floating-Point Add, except that the contents of register **fr**B participates 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. **fsub** Floating-Point Subtract **fsub.** Floating-Point Subtract with CR Update. The dot suffix enables the update of the condition register.
Floating-Point Subtract Single-Precision	**fsubs** **fsubs.**	**fr**D,**fr**A,**fr**B	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 the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **fr**D. The execution of the Floating-Point Subtract instruction is identical to that of Floating-Point Add, except that the contents of register **fr**B participates 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. **fsubs** Floating-Point Subtract Single-Precision **fsubs.** Floating-Point Subtract Single-Precision with CR Update. The dot suffix enables the update of the condition register.

Table 4-7 Floating-Point Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Multiply	**fmul** **fmul.**	**fr**D,**fr**A,**fr**C	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 the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **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. **fmul** Floating-Point Multiply **fmul.** Floating-Point Multiply with CR Update. The dot suffix enables the update of the condition register.
Floating-Point Multiply Single-Precision	**fmuls** **fmuls.**	**fr**D,**fr**A,**fr**C	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 the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **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. **fmuls** Floating-Point Multiply Single-Precision **fmuls.** Floating-Point Multiply Single-Precision with CR Update. The dot suffix enables the update of the condition register.
Floating-Point Divide	**fdiv** **fdiv.**	**fr**D,**fr**A,**fr**B	The floating-point operand in register **fr**A is divided by the floating-point operand in register **fr**B. No remainder is preserved. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **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. **fdiv** Floating-Point Divide **fdiv.** Floating-Point Divide with CR Update. The dot suffix enables the update of the condition register.

Table 4-7 Floating-Point Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Divide Single-Precision	**fdivs** **fdivs.**	**fr**D,**fr**A,**fr**B	The floating-point operand in register **fr**A is divided by the floating-point operand in register **fr**B. No remainder is preserved. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **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. **fdivs** Floating-Point Divide Single-Precision **fdivs.** Floating-Point Divide Single-Precision with CR Update. The dot suffix enables the update of the condition register.

4.4.2 Floating-Point Multiply-Add Instructions

These instructions combine multiply and add operations without an intermediate rounding operation. The fractional part of the intermediate product is 106 bits wide, and all 106 bits take part in the add/subtract portion of the instruction.

The floating-point multiply-add instructions are summarized in **Table 4-8**.

Table 4-8 Floating-Point Multiply-Add Instructions

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Multiply-Add	**fmadd** **fmadd.**	**fr**D,**fr**A,**fr**C,**fr**B	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 the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **fr**D. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1. **fmadd** Floating-Point Multiply-Add **fmadd.** Floating-Point Multiply-Add with CR Update. The dot suffix enables the update of the condition register.

Table 4-8 Floating-Point Multiply-Add Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Multiply-Add Single-Precision	**fmadds** **fmadds.**	**fr**D,**fr**A,**fr**C,**fr**B	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 the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **fr**D. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1. **fmadds** Floating-Point Multiply-Add Single-Precision **fmadds.** Floating-Point Multiply-Add Single-Precision with CR Update. The dot suffix enables the update of the condition register.
Floating-Point Multiply-Subtract	**fmsub** **fmsub.**	**fr**D,**fr**A,**fr**C,**fr**B	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 the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **fr**D. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1. **fmsub** Floating-Point Multiply-Subtract **fmsub.** Floating-Point Multiply-Subtract with CR Update. The dot suffix enables the update of the condition register.
Floating-Point Multiply-Subtract Single-Precision	**fmsubs** **fmsubs.**	**fr**D,**fr**A,**fr**C,**fr**B	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 the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **fr**D. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1. **fmsubs** Floating-Point Multiply-Subtract Single-Precision **fmsubs.** Floating-Point Multiply-Subtract Single-Precision with CR Update. The dot suffix enables the update of the condition register.

Table 4-8 Floating-Point Multiply-Add Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Negative Multiply-Add	**fnmadd** **fnmadd.**	**fr**D,**fr**A,**fr**C,**fr**B	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 the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into register **fr**D. This instruction produces the same result as would be obtained by using the floating-point multiply-add instruction and then negating the result, with the following exceptions: • QNaNs propagate with no effect on their sign bit. • QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of 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. **fnmadd** Floating-Point Negative Multiply-Add **fnmadd.** Floating-Point Negative Multiply-Add with CR Update. The dot suffix enables the update of the condition register.
Floating-Point Negative Multiply-Add Single-Precision	**fnmadds** **fnmadds.**	**fr**D,**fr**A,**fr**C,**fr**B	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 the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into register **fr**D. This instruction produces the same result as would be obtained by using the floating-point multiply-add instruction and then negating the result, with the following exceptions: • QNaNs propagate with no effect on their sign bit. • QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of 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. **fnmadds** Floating-Point Negative Multiply-Add Single-Precision **fnmadds.** Floating-Point Negative Multiply-Add Single-Precision with CR Update. The dot suffix enables the update of the condition register.

Table 4-8 Floating-Point Multiply-Add Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Negative Multiply-Subtract	**fnmsub** **fnmsub.**	**fr**D,**fr**A,**fr**C,**fr**B	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 the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into register **fr**D. This instruction produces the same result as would be obtained by using the floating-point multiply-subtract instruction and then negating the result, with the following exceptions: • QNaNs propagate with no effect on their sign bit. • QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of 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. **fnmsub** Floating-Point Negative Multiply-Subtract **fnmsub.** Floating-Point Negative Multiply-Subtract with CR Update. The dot suffix enables the update of the condition register.
Floating-Point Negative Multiply-Subtract Single-Precision	**fnmsubs** **fnmsubs.**	**fr**D,**fr**A,**fr**C,**fr**B	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 the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into register **fr**D. This instruction produces the same result as would be obtained by using the floating-point multiply-subtract instruction and then negating the result, with the following exceptions: • QNaNs propagate with no effect on their sign bit. • QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of 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. **fnmsubs** Floating-Point Negative Multiply-Subtract Single-Precision **fnmsubs.** Floating-Point Negative Multiply-Subtract Single-Precision with CR Update. The dot suffix enables the update of the condition register.

4.4.3 Floating-Point Rounding and Conversion Instructions

The floating-point rounding instruction is used to produce a 32-bit single-precision number from a 64-bit double-precision floating-point number. The floating-point convert instructions convert 64-bit double-precision floating point numbers to 32-bit signed integer numbers.

Examples of uses of these instructions to perform various conversions can be found in **APPENDIX C FLOATING-POINT MODELS AND CONVERSIONS**.

Table 4-9 Floating-Point Rounding and Conversion Instructions

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Round to Single-Precision	**frsp** **frsp.**	**fr**D,**fr**B	If it is already in single-precision range, the floating-point operand in register **fr**B is placed into register **fr**D. Otherwise the floating-point operand in register **fr**B is rounded to single-precision using the rounding mode specified by FPSCR[RN] and placed into register **fr**D. The rounding is described fully in **APPENDIX C FLOATING-POINT MODELS AND CONVERSIONS**. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1. **frsp** Floating-Point Round to Single-Precision **frsp.** Floating-Point Round to Single-Precision with CR Update. The dot suffix enables the update of the condition register.
Floating-Point Convert to Integer Word	**fctiw** **fctiw.**	**fr**D,**fr**B	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 **fr**D[32:63]. **fr**D[0:31] are undefined. If the operand in register **fr**B is greater than $2^{31}-1$, **fr**D[32:63] are set to 0x7FFF FFFF. If the operand in register **fr**B is less than -2^{31}, **fr**D[32:63] are set to 0x8000 0000. The conversion is described fully in **APPENDIX C FLOATING-POINT MODELS AND CONVERSIONS**. 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. **fctiw** Floating-Point Convert to Integer Word **fctiw.** Floating-Point Convert to Integer Word with CR Update. The dot suffix enables the update of the condition register.

Table 4-9 Floating-Point Rounding and Conversion Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Convert to Integer Word with Round	**fctiwz** **fctiwz.**	**fr**D,**fr**B	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 **fr**D[32:63]. **fr**D[0:31] are undefined. If the operand in **fr**B is greater than $2^{31}-1$, **fr**D[32:63] are set to 0x7FFF FFFF. If the operand in register **fr**B is less than -2^{31}, **fr**D[32:63] are set to 0x8000 0000. The conversion is described fully in **APPENDIX C FLOATING-POINT MODELS AND CONVERSIONS**. 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. **fctiwz** Floating-Point Convert to Integer Word with Round Toward Zero **fctiwz.** Floating-Point Convert to Integer Word with Round Toward Zero with CR Update. The dot suffix enables the update of the condition register.

4.4.4 Floating-Point Compare Instructions

Floating-point compare instructions compare the contents of two floating-point registers and the comparison ignores the sign of zero (that is +0 = –0). The comparison can be ordered or unordered. The comparison sets one bit in the designated CR field and clears the other three bits. The FPCC bits (FPSCR[16:19]) are set in the same way.

The CR field and the FPCC are interpreted as shown in **Table 4-10**.

Table 4-10 CR Bit Settings

Bit	Name	Description
0	FL	(**fr**A) < (**fr**B)
1	FG	(**fr**A) > (**fr**B)
2	FE	(**fr**A) = (**fr**B)
3	FU	(**fr**A) ? (**fr**B) (unordered)

On floating-point compare unordered (**fcmpu**) and floating-point compare ordered (**fcmpo**) instructions with condition register updating enabled, the PowerPC architecture defines CR1 and the CR field specified by operand **crf**D as undefined.

The floating-point compare instructions are summarized in **Table 4-11**.

Table 4-11 Floating-Point Compare Instructions

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Compare Unordered	**fcmpu**	**crf**D,**fr**A,**fr**B	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 an operand is a NaN, either quiet or signaling, CR field **crf**D and the FPCC are set to reflect unordered. If an operand is a Signaling NaN, VXSNAN is set.
Floating-Point Compare Ordered	**fcmpo**	**crf**D,**fr**A,**fr**B	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 an operand is a NaN, either quiet or signaling, CR field **crf**D and the FPCC are set to reflect unordered. If an operand is a Signaling NaN, VXSNAN is set, and if invalid operation is disabled (VE = 0) then VXVC is set. Otherwise, if an operand is a Quiet NaN, VXVC is set.

4.4.5 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. In particular:

- All exceptions caused by the previously initiated instructions are recorded in the FPSCR before the FPSCR instruction is initiated.
- All invocations of the floating-point exception handler caused by the previously initiated instructions have occurred before the FPSCR instruction is initiated.
- No subsequent floating-point instruction that depends on or alters the settings of any FPSCR bits appears to be initiated until the FPSCR instruction has completed.

Floating-point memory access instructions are not affected.

The floating-point status and control register instructions are summarized in **Table 4-12**.

Table 4-12 Floating-Point Status and Control Register Instructions

Name	Mnemonic	Operand Syntax	Operation
Move from FPSCR	**mffs** **mffs.**	**fr**D	The contents of the FPSCR are placed into **fr**D[32:63]. **mffs** Move from FPSCR **mffs.** Move from FPSCR with CR Update. The dot suffix enables the update of the condition register.
Move to Condition Register from FPSCR	**mcrfs**	**crf**D,**crf**S	The contents of FPSCR field specified by operand **crf**S are copied to the CR field specified by operand **crf**D. All exception bits copied are cleared to zero in the FPSCR.
Move to FPSCR Field Immediate	**mtfsfi** **mtfsfi.**	**crf**D,IMM	The value of the IMM field is placed into FPSCR field **crf**D. All other FPSCR fields are unchanged. **mtfsfi** Move to FPSCR Field Immediate **mtfsfi.** Move to FPSCR Field Immediate with CR Update. The dot suffix enables the update of the condition register. When FPSCR[0:3] is specified, bits 0 (FX) and 3 (OX) are set to the values of IMM[0] and IMM[3] (i.e., even if this instruction causes OX to change from zero to one, FX is set from IMM[0] and not by the usual rule that FX is set to one when an exception bit changes from zero to one). Bits 1 and 2 (FEX and VX) are set according to the usual rule described in **2.2.3 Floating-Point Status and Control Register (FPSCR)**, and not from IMM[1:2].
Move to FPSCR Fields	**mtfsf** **mtfsf.**	FM,**fr**B	**fr**B[32:63] 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 = 1 then 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. **mtfsf** Move to FPSCR Fields **mtfsf.** Move to FPSCR Fields with CR Update. The dot suffix enables the update of the condition register. 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] (i.e., even if this instruction causes OX to change from zero to one, FX is set from **fr**B[32] and not by the usual rule that FX is set to one when an exception bit changes from zero to one). Bits 1 and 2 (FEX and VX) are set according to the usual rule described in **2.2.3 Floating-Point Status and Control Register (FPSCR)**, and not from **fr**B[33:34].
Move to FPSCR Bit 0	**mtfsb0** **mtfsb0.**	**crb**D	The bit of the FPSCR specified by operand **crb**D is cleared to zero. Bits 1 and 2 (FEX and VX) cannot be explicitly reset. **mtfsb0** Move to FPSCR Bit 0 **mtfsb0.** Move to FPSCR Bit 0 with CR Update. The dot suffix enables the update of the condition register.
Move to FPSCR Bit 1	**mtfsb1** **mtfsb1.**	**crb**D	The bit of the FPSCR specified by operand **crb**D is set to one. Bits 1 and 2 (FEX and VX) cannot be reset explicitly. **mtfsb1** Move to FPSCR Bit 1 **mtfsb1.** Move to FPSCR Bit 1 with CR Update. The dot suffix enables the update of the condition register.

4.5 Load and Store Instructions

The RCPU supports the following types of load and store instructions:

- Integer load instructions
- Integer store instructions
- Integer load and store with byte reversal instructions
- Integer load and store multiple instructions
- Floating-point load instructions
- Floating-point store instructions
- Floating-point move instructions
- Memory synchronization instructions (described in **4.8 Memory Synchronization Instructions**)

4.5.1 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.

4.5.1.1 Register Indirect with Immediate Index Addressing

Instructions using this addressing mode contain a signed 16-bit immediate index (d operand) which is sign extended to 32 bits, and added to the contents of a general purpose register specified in the instruction (**r**A operand) to generate the effective address. A zero in place of the **r**A operand causes a zero to be added to the immediate index (d operand). The option to specify **r**A or zero is shown in the instruction descriptions as (**r**A|0).

Figure 4-1 shows how an effective address is generated when using register indirect with immediate index addressing.

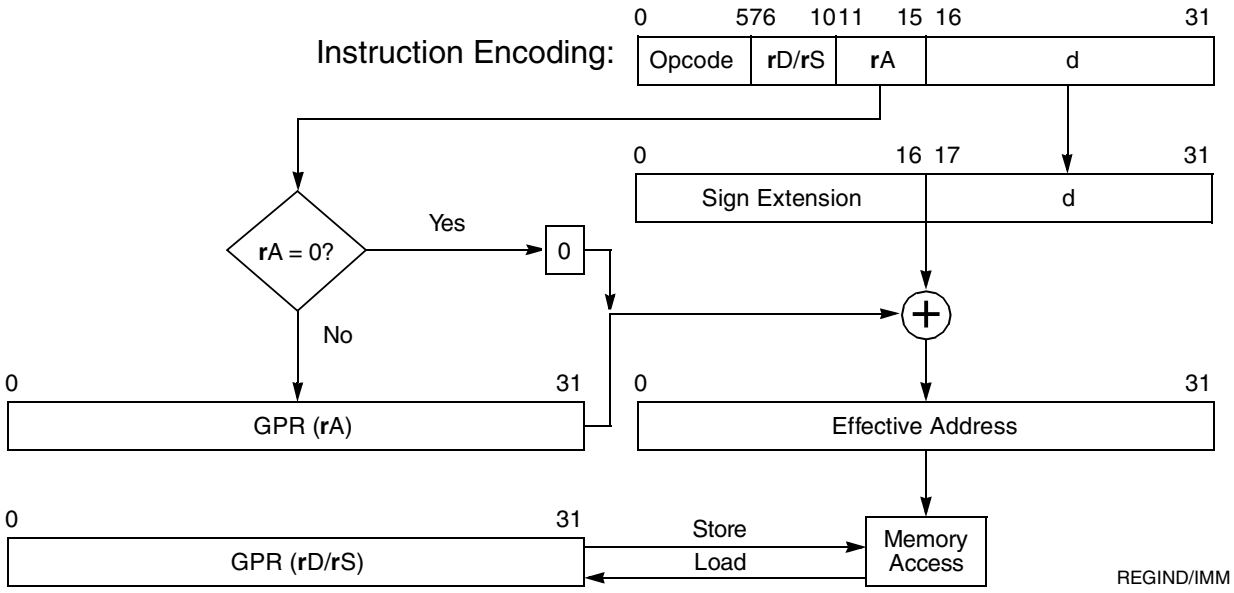

Figure 4-1 Register Indirect with Immediate Index Addressing

4.5.1.2 Register Indirect with Index Addressing

Instructions using this addressing mode cause the contents of two general purpose registers (specified as operands **r**A and **r**B) to be added in the generation of the effective address. A zero in place of the **r**A operand causes a zero to be added to the contents of the general-purpose register specified in operand **r**B. The option to specify **r**A or zero is shown in the instruction descriptions as (**r**A|0).

Figure 4-2 shows how an effective address is generated when using register indirect with index addressing.

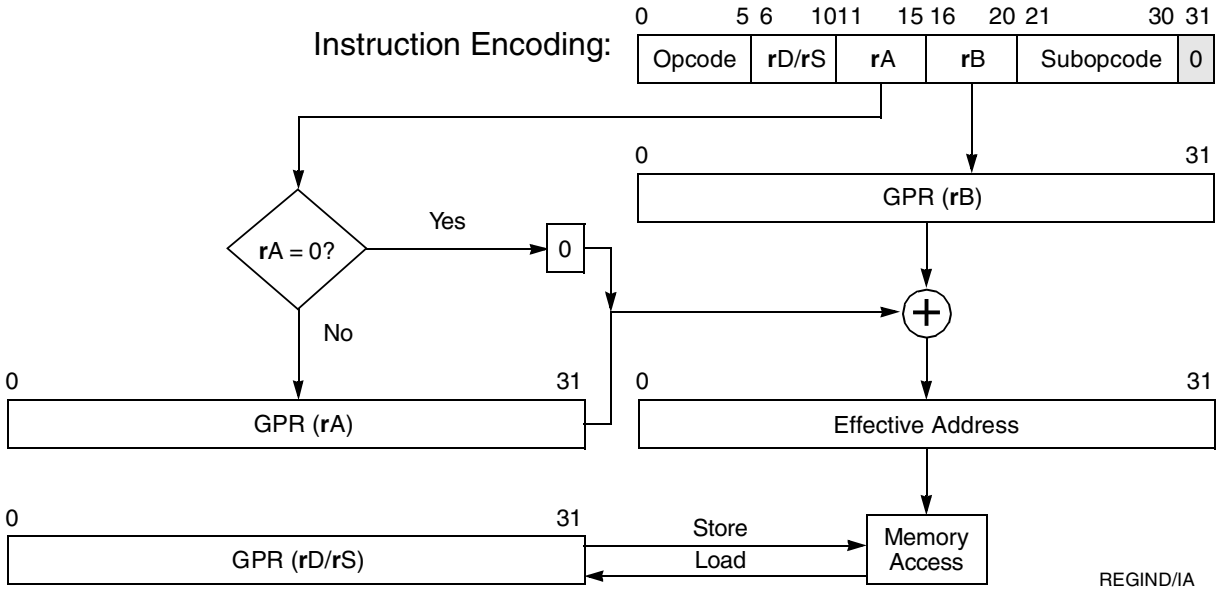

Figure 4-2 Register Indirect with Index Addressing

4.5.1.3 Register Indirect Addressing

Instructions using this addressing mode use the contents of the general purpose register specified by the **r**A operand as the effective address. A zero in the **r**A operand causes an effective address of zero to be generated. The option to specify **r**A or zero is shown in the instruction descriptions as (**r**A|0).

Figure 4-3 shows how an effective address is generated when using register indirect addressing.

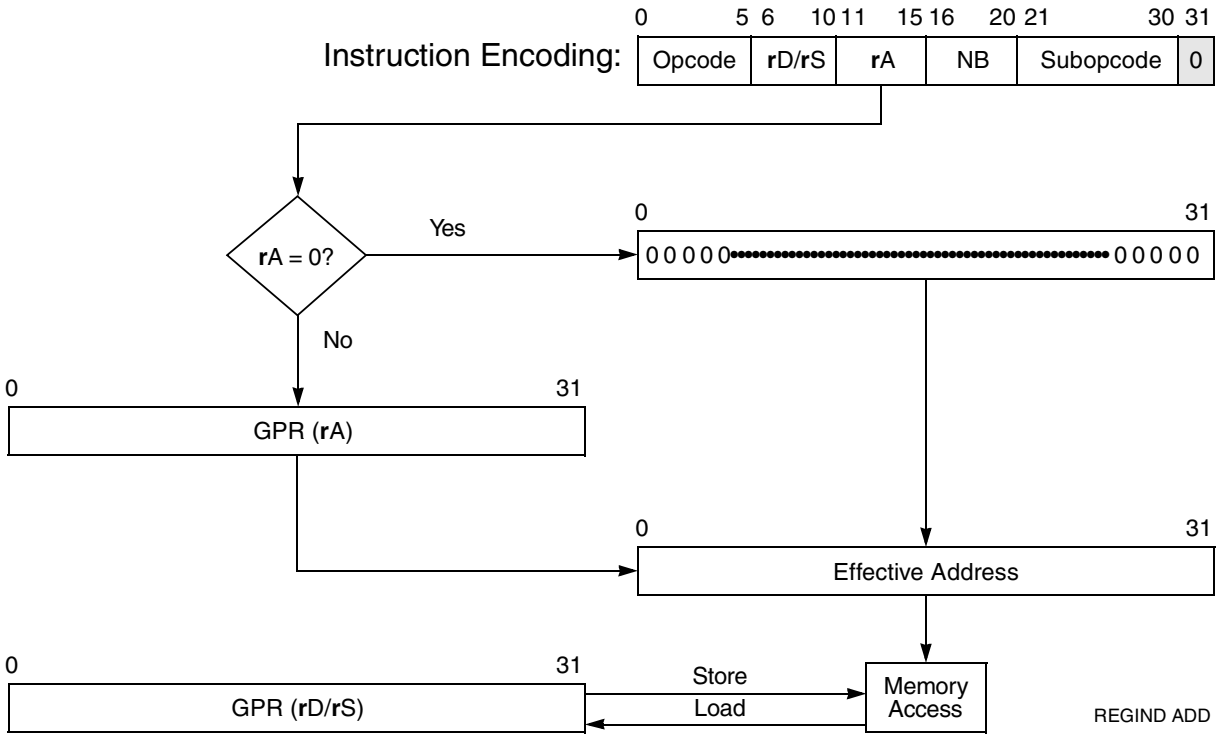

Figure 4-3 Register Indirect Addressing

4.5.2 Integer Load Instructions

For load instructions, the byte, half-word, word, or double-word addressed by EA 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, the effective address is placed into **r**A and the memory element (byte, half-word, or word) addressed by EA is loaded into **r**D.

The PowerPC architecture defines load with update instructions with **r**A = 0 or **r**A = **r**D as an invalid form. In the RCPU, however, if **r**A = 0 then the EA is written into R0. If **r**A = **r**D then **r**A is loaded from memory location MEM(**r**A, N) where N is determined by the instruction operand size.

Table 4-13 summarizes the RCPU load instructions.

Table 4-13 Integer Load Instructions

Name	Mnemonic	Operand Syntax	Operation
Load Byte and Zero	**lbz**	**r**D,**d(r**A)	The effective address is the sum (**r**A|0) + d. The byte in memory addressed by the EA is loaded into register **r**D[24:31]. The remaining bits in register **r**D are cleared to zero.
Load Byte and Zero Indexed	**lbzx**	**r**D,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). The byte in memory addressed by the EA is loaded into register **r**D[24:31]. The remaining bits in register **r**D are cleared to zero.
Load Byte and Zero with Update	**lbzu**	**r**D,**d(r**A)	The effective address (EA) is the sum (**r**A|0) + d. The byte in memory addressed by the EA is loaded into register **r**D[24:31]. The remaining bits in register **r**D are cleared to zero. The EA is placed into register **r**A. The PowerPC architecture defines load with update instructions with **r**A = 0 or **r**A = **r**D as invalid forms. In the RCPU, however, if **r**A = 0 then the EA is written into R0. If **r**A = **r**D then **r**A is loaded from memory location MEM(**r**A, N) where N is determined by the instruction operand size.
Load Byte and Zero with Update Indexed	**lbzux**	**r**D,**r**A,**r**B	The effective address (EA) is the sum (**r**A|0) + (**r**B). The byte addressed by the EA is loaded into register **r**D[24:31]. The remaining bits in register **r**D are cleared to zero. The EA is placed into register **r**A. The PowerPC architecture defines load with update instructions with **r**A = 0 or **r**A = **r**D as invalid forms. In the RCPU, however, if **r**A = 0 then the EA is written into R0. If **r**A = **r**D then **r**A is loaded from memory location MEM(**r**A, N) where N is determined by the instruction operand size.
Load Half Word and Zero	**lhz**	**r**D,**d(r**A)	The effective address is the sum (**r**A|0) + d. The half-word in memory addressed by the EA is loaded into register **r**D[16:31]. The remaining bits in **r**D are cleared to zero.
Load Half Word and Zero Indexed	**lhzx**	**r**D,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). The half-word in memory addressed by the EA is loaded into register **r**D[16:31]. The remaining bits in register **r**D are cleared.
Load Half Word and Zero with Update	**lhzu**	**r**D,**d(r**A)	The effective address is the sum (**r**A|0) + d. The half-word in memory addressed by the EA is loaded into register **r**D[16:31]. The remaining bits in register **r**D are cleared. The EA is placed into register **r**A. The PowerPC architecture defines load with update instructions with **r**A = 0 or **r**A =**r**D as invalid forms. In the RCPU, however, if **r**A=0 then the EA is written into R0. If **r**A = **r**D then **r**A is loaded from memory location MEM(**r**A, N) where N is determined by the instruction operand size.

Table 4-13 Integer Load Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Load Half Word and Zero with Update Indexed	**lhzux**	**r**D,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). The half-word in memory addressed by the EA is loaded into register **r**D[16:31]. The remaining bits in register **r**D are cleared. The EA is placed into register **r**A. The PowerPC architecture defines load with update instructions with **r**A = 0 or **r**A = **r**D as invalid forms. In the RCPU, however, if **r**A = 0 then the EA is written into R0. If **r**A = **r**D then **r**A is loaded from memory location MEM(**r**A, N) where N is determined by the instruction operand size.
Load Half Word Algebraic	**lha**	**r**D,**d(r**A**)**	The effective address is the sum (**r**A) + d. The half-word in memory addressed by the EA is loaded into register **r**D[16:31]. The remaining bits in register **r**D are filled with a copy of bit 0 of the loaded half-word.
Load Half Word Algebraic Indexed	**lhax**	**r**D,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). The half-word in memory addressed by the EA is loaded into register **r**D[16:31]. The remaining bits in register **r**D are filled with a copy of bit 0 of the loaded half-word.
Load Half Word Algebraic with Update	**lhau**	**r**D,**d(r**A**)**	The effective address is the sum (**r**A|0) + d. The half-word in memory addressed by the EA is loaded into register **r**D[16:31]. The remaining bits in register **r**D are filled with a copy of bit 0 of the loaded half-word. The EA is placed into register **r**A. The PowerPC architecture defines load with update instructions with **r**A = 0 or **r**A = **r**D as invalid forms. In the RCPU, however, if **r**A = 0 then the EA is written into R0. If **r**A = **r**D then **r**A is loaded from memory location MEM(**r**A, N) where N is determined by the instruction operand size.
Load Half Word Algebraic with Update Indexed	**lhaux**	**r**D,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). The half-word in memory addressed by the EA is loaded into register **r**D[16:31]. The remaining bits in register **r**D are filled with a copy of bit 0 of the loaded half-word. The EA is placed into register **r**A. The PowerPC architecture defines load with update instructions with **r**A = 0 or **r**A = **r**D as invalid forms. In the RCPU, however, if **r**A = 0 then the EA is written into R0. If **r**A = **r**D then **r**A is loaded from memory location MEM(**r**A, N) where N is determined by the instruction operand size.
Load Word and Zero	**lwz**	**r**D,**d(r**A**)**	The effective address is the sum (**r**A|0) + d. The word in memory addressed by the EA is loaded into register **r**D[0:31].
Load Word and Zero Indexed	**lwzx**	**r**D,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). The word in memory addressed by the EA is loaded into register **r**D[0:31].
Load Word and Zero with Update	**lwzu**	**r**D,**d(r**A**)**	The effective address is the sum (**r**A|0) + d. The word in memory addressed by the EA is loaded into register **r**D[0:31]. The EA is placed into register **r**A. The PowerPC architecture defines load with update instructions with **r**A = 0 or **r**A = **r**D as invalid forms. In the RCPU, however, if **r**A = 0 then the EA is written into R0. If **r**A = **r**D then **r**A is loaded from memory location MEM(**r**A, N) where N is determined by the instruction operand size.

Table 4-13 Integer Load Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Load Word and Zero with Update Indexed	**lwzux**	**r**D,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). The word in memory addressed by the EA is loaded into register **r**D[0:31]. The EA is placed into register **r**A. The PowerPC architecture defines load with update instructions with **r**A = 0 or **r**A = **r**D as invalid forms. In the RCPU, however, if **r**A = 0 then the EA is written into R0. If **r**A = **r**D then **r**A is loaded from memory location MEM(**r**A, N) where N is determined by the instruction operand size.

4.5.3 Integer Store Instructions

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

- If **r**A ¦ 0, the effective address is placed into register **r**A.
- If **r**A = 0, the effective address is written into R0. (Although the PowerPC architecture defines store with update instructions with **r**A = 0 as invalid forms, the RCPU does not.)
- 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.

A summary of the RCPU integer store instructions is shown in **Table 4-14**.

Table 4-14 Integer Store Instructions

Name	Mnemonic	Operand Syntax	Operation
Store Byte	**stb**	**r**S,**d(r**A)	The effective address is the sum (**r**A|0) + d. Register **r**S[24:31] is stored into the byte in memory addressed by the EA.
Store Byte Indexed	**stbx**	**r**S,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). **r**S[24:31] is stored into the byte in memory addressed by the EA.
Store Byte with Update	**stbu**	**r**S,**d(r**A)	The effective address is the sum (**r**A|0) + d. **r**S[24:31] is stored into the byte in memory addressed by the EA. The EA is placed into register **r**A.
Store Byte with Update Indexed	**stbux**	**r**S,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). **r**S[24:31] is stored into the byte in memory addressed by the EA. The EA is placed into register **r**A.
Store Half Word	**sth**	**r**S,**d(r**A)	The effective address is the sum (**r**A|0) + d. **r**S[16:31] is stored into the half-word in memory addressed by the EA.
Store Half Word Indexed	**sthx**	**r**S,**r**A,**r**B	The effective address (EA) is the sum (**r**A|0) + (**r**B). **r**S[16:31] is stored into the half-word in memory addressed by the EA.
Store Half Word with Update	**sthu**	**r**S,**d(r**A)	The effective address is the sum (**r**A|0) + d. **r**S[16:31] is stored into the half-word in memory addressed by the EA. The EA is placed into register **r**A.
Store Half Word with Update Indexed	**sthux**	**r**S,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). **r**S[16:31] is stored into the half-word in memory addressed by the EA. The EA is placed into register **r**A.
Store Word	**stw**	**r**S,**d(r**A)	The effective address is the sum (**r**A|0) + d. Register **r**S is stored into the word in memory addressed by the EA.
Store Word Indexed	**stwx**	**r**S,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). **r**S is stored into the word in memory addressed by the EA.
Store Word with Update	**stwu**	**r**S,**d(r**A)	The effective address is the sum (**r**A|0) + d. Register **r**S is stored into the word in memory addressed by the EA. The EA is placed into register **r**A.
Store Word with Update Indexed	**stwux**	**r**S,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). Register **r**S is stored into the word in memory addressed by the EA. The EA is placed into register **r**A.

4.5.4 Integer Load and Store with Byte Reversal Instructions

Table 4-15 describes the integer load and store with byte reversal instructions.

Table 4-15 Integer Load and Store with Byte Reversal Instructions

Name	Mnemonic	Operand Syntax	Operation
Load Half Word Byte-Reverse Indexed	**lhbrx**	**r**D,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). Bits 0 to 7 of the half-word in memory addressed by the EA are loaded into **r**D[24:31]. Bits 8 to 15 of the half-word in memory addressed by the EA are loaded into **r**D[16:23]. The rest of the bits in **r**D are cleared to zero.
Load Word Byte-Reverse Indexed	**lwbrx**	**r**D,**r**A,**r**B	The effective address is the sum (**r**A|0)+(**r**B). Bits 0–7 of the word in memory addressed by the EA are loaded into **r**D[24:31]. Bits 8 to 15 of the word in memory addressed by the EA are loaded into **r**D[16:23]. Bits 16 to 23 of the word in memory addressed by the EA are loaded into **r**D[8:15]. Bits 24 to 31 of the word in memory addressed by the EA are loaded into **r**D[0:7].
Store Half Word Byte-Reverse Indexed	**sthbrx**	**r**S,**r**A,**r**B	The effective address is the sum (**r**A|0)+(**r**B). **r**S[24:31] are stored into bits 0 to 7 of the half-word in memory addressed by the EA. **r**S[16:23] are stored into bits 8 to 15 of the half-word in memory addressed by the EA.
Store Word Byte-Reverse Indexed	**stwbrx**	**r**S,**r**A,**r**B	The effective address is the sum (**r**A|0)+(**r**B). **r**S[24:31] are stored into bits 0 to 7 of the word in memory addressed by EA. Register **r**S[16:23] are stored into bits 8 to 15 of the word in memory addressed by the EA. Register **r**S[8:15] are stored into bits 16 to 23 of the word in memory addressed by the EA. **r**S[0:7] are stored into bits 24 to 31 of the word in memory addressed by the EA.

4.5.5 Integer Load and Store Multiple Instructions

The load/store multiple instructions are used to move blocks of data to and from the GPRs.

The PowerPC architecture defines the load multiple instruction (**lmw**) with **r**A in the range of registers to be loaded as an invalid form. In the RCPU, however, if **r**A is in the range of registers to be loaded, the instruction completes normally, and **r**A is loaded from the memory location as follows:

rA ← MEM(EA+(**r**A–**r**S)*4, 4)

For integer load and store multiple instructions, the effective address must be a multiple of four. If not, a system alignment exception is generated.

Table 4-16 Integer Load and Store Multiple Instructions

Name	Mnemonic	Operand Syntax	Operation
Load Multiple Word	**lmw**	**r**D,d(**r**A)	The effective address is the sum (**r**A|0)+d. *n* = 32 –**r**D. *n* consecutive words starting at EA are loaded into GPRs **r**D through 31. If the EA is not a multiple of four the alignment exception handler is invoked.
Store Multiple Word	**stmw**	**r**S,d(**r**A)	The effective address is the sum (**r**A|0)+d. *n* = (32 –**r**S). *n* consecutive words starting at the EA are stored from GPRs **r**S through 31. If the EA is not a multiple of four the alignment exception handler is invoked.

4.5.6 Integer Move String Instructions

The integer move 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.

Load/store string indexed instructions of zero length have no effect, except that load string indexed instructions of zero length may set register **r**D to an undefined value.

The PowerPC architecture defines the load string instructions with **r**A in the range of registers to be loaded as an invalid form. In the RCPU, however, if **r**A is in the range of registers to be loaded, the instruction completes normally, and **r**A is loaded from memory.

Table 4-17 Integer Move String Instructions

Name	Mnemonic	Operand Syntax	Operation
Load String Word Immediate	**lswi**	**r**D,**r**A,NB	The 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* = (*n*/4); *nr* is the number of registers to receive data. *n* consecutive bytes starting at the 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 four bytes of register **r**D + *nr* - 1 are only partially filled, the unfilled low-order byte(s) of that register are cleared to zero. The PowerPC architecture defines the load string instructions with **r**A in the range of registers to be loaded as an invalid form. In the RCPU, however, if **r**A is in the range of registers to be loaded, the instruction completes normally, and **r**A is loaded from memory.
Load String Word Indexed	**lswx**	**r**D,**r**A,**r**B	The 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 the EA are loaded into registers **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 four bytes of register **r**D + *nr* - 1 are only partially filled, the unfilled low-order byte(s) of that register are cleared to zero. If *n* = 0, the contents of register **r**D is undefined. The PowerPC architecture defines the load string instructions with **r**A in the range of registers to be loaded as an invalid form. In the RCPU, however, if **r**A is in the range of registers to be loaded, the instruction completes normally, and **r**A is loaded from memory.
Store String Word Immediate	**stswi**	**r**S,**r**A,NB	The EA is (**r**A|0). Let *n* = NB if NB¦0, *n* = 32 if NB = 0; *n* is the number of bytes to store. Let *nr* = CEIL(*n*/4); *nr* is the number of registers to supply data. *n* consecutive bytes starting at the EA are stored from register **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.
Store String Word Indexed	**stswx**	**r**S,**r**A,**r**B	The effective address 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 the EA are stored from register **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.

4.5.7 Floating-Point Load and Store Address Generation

Floating point load and store operations generate effective addresses using the register indirect with immediate index mode and register indirect with index mode, the details of which are described below.

4.5.7.1 Register Indirect with Immediate Index Addressing

Instructions using this addressing mode contain a signed 16-bit immediate index (d operand) which is sign extended to 32 bits, and added to the contents of a general purpose register specified in the instruction (**r**A operand) to generate the effective address. A zero in the **r**A operand causes a zero to be added to the immediate index (d operand). This is shown in the instruction descriptions as (**r**A|0).

Figure 4-4 shows how an effective address is generated when using register indirect with immediate index addressing.

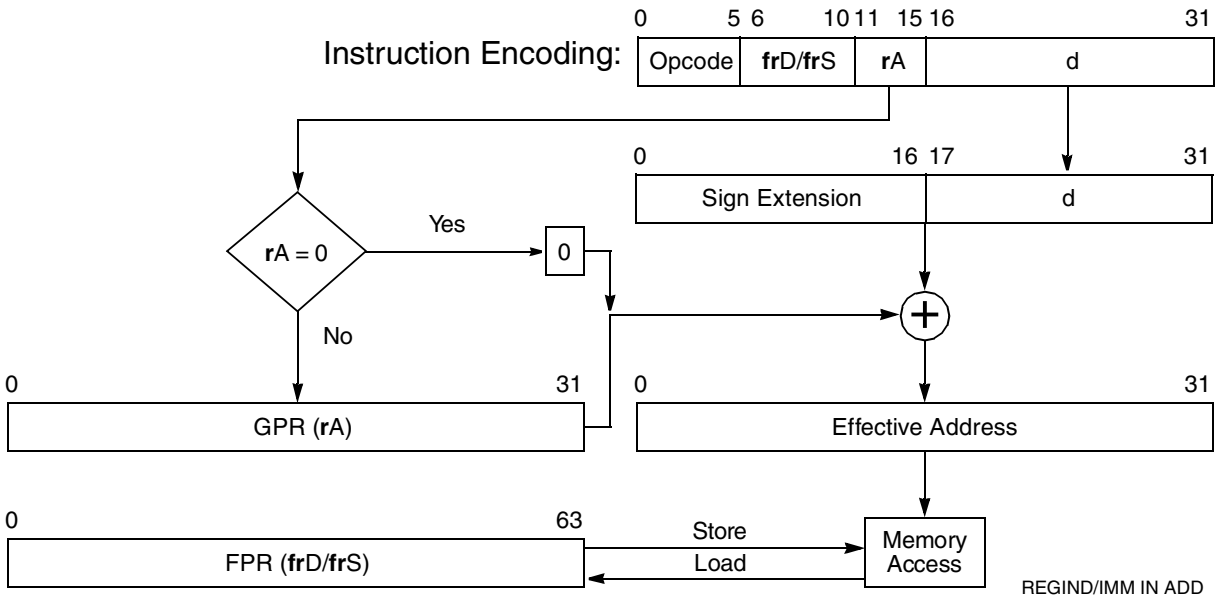

Figure 4-4 Register Indirect with Immediate Index Addressing

4.5.7.2 Register Indirect with Index Addressing

Instructions using this addressing mode add the contents of two general-purpose registers (specified in operands **r**A and **r**B) to generate the effective address. A zero in the **r**A operand causes a zero to be added to the contents of general-purpose register specified in operand **r**B. This is shown in the instruction descriptions as (**r**A|0).

Figure 4-5 shows how an effective address is generated when using register indirect with index addressing.

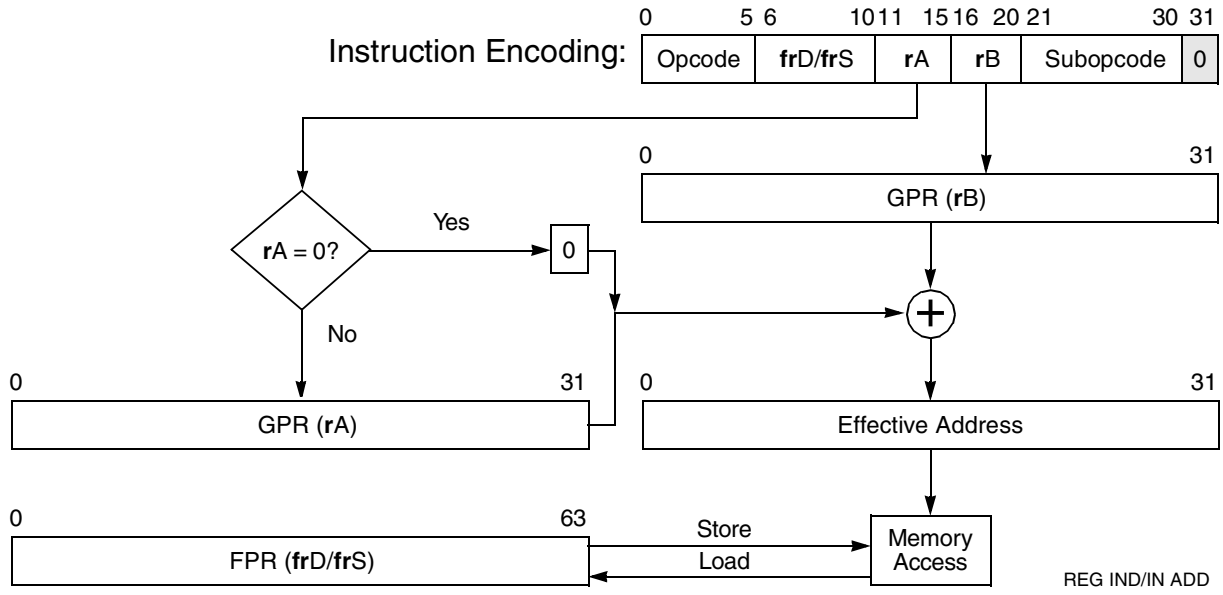

Figure 4-5 Register Indirect with Index Addressing

4.5.8 Floating-Point Load Instructions

There are two basic forms of floating-point load instruction: single-precision and double-precision formats. Because the FPRs support only floating-point, double-precision format, single-precision floating-point load instructions convert single-precision data to double-precision format before loading the operands into the target FPR. This conversion is described in **4.5.8.1 Double-Precision Conversion for Floating-Point Load Instructions**. **Table 4-18** provides a summary of the floating-point load instructions.

Table 4-18 Floating-Point Load Instructions

Name	Mnemonic	Operand Syntax	Operation
Load Floating-Point Single-Precision	**lfs**	**fr**D,d(**r**A)	The effective address is the sum (**r**A|0)+d. The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision format and placed into register **fr**D.
Load Floating-Point Single-Precision Indexed	**lfsx**	**fr**D,**r**A,**r**B	The effective address is the sum (**r**A|0)+(**r**B). The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision and placed into register **fr**D.

Table 4-18 Floating-Point Load Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Load Floating-Point Single-Precision with Update	**lfsu**	**fr**D,d(**r**A)	The effective address is the sum (**r**A|0)+d. The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see **4.5.8.1 Double-Precision Conversion for Floating-Point Load Instructions**) and placed into register **fr**D. The EA is placed into the register specified by **r**A.
Load Floating-Point Single-Precision with Update Indexed	**lfsux**	**fr**D,**r**A,**r**B	The effective address is the sum (**r**A|0)+(**r**B). The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see **4.5.8.1 Double-Precision Conversion for Floating-Point Load Instructions**) and placed into register **fr**D. The EA is placed into the register specified by **r**A.
Load Floating-Point Double-Precision	**lfd**	**fr**D,d(**r**A)	The effective address is the sum (**r**A|0)+d. The double-word in memory addressed by the EA is placed into register **fr**D.
Load Floating-Point Double-Precision Indexed	**lfdx**	**fr**D,**r**A,**r**B	The effective address is the sum (**r**A|0)+(**r**B). The double-word in memory addressed by the EA is placed into register **fr**D.
Load Floating-Point Double-Precision with Update	**lfdu**	**fr**D,d(**r**A)	The effective address is the sum (**r**A|0)+d. The double-word in memory addressed by the EA is placed into register **fr**D. The EA is placed into the register specified by **r**A.
Load Floating-Point Double-Precision with Update Indexed	**lfdux**	**fr**D,**r**A,**r**B	The effective address is the sum (**r**A|0)+(**r**B). The double-word in memory addressed by the EA is placed into register **fr**D. The EA is placed into the register specified by **r**A.

4.5.8.1 Double-Precision Conversion for Floating-Point Load Instructions

The steps for converting from single- to double-precision and loading are as follows:

WORD[0:31] is the floating-point, single-precision operand accessed from memory.

Normalized Operand

If WORD[1:8] >0 and WORD[1:8]<255
    frD[0:1] < WORD[0:1]
    frD[2] < ¬WORD[1]
    frD[3] < ¬WORD[1]
    frD[4]< ¬WORD[1]
    frD[5:63] < WORD[2:31] || ^{290}b0

Denormalized Operand

If WORD[1:8] =0 and WORD[9:31] ¦0
    sign < WORD[0]
    exp < -126
    frac[0:52] < 0b0 || WORD[9:31] || ^{2}0b0
    normalize the operand
        Do while frac 0 =0
            frac < frac[1:52] || 0b0
            exp < exp - 1
        End
    frD[0] < sign
    frD[1:11] < exp + 1023
    frD[12:63] < frac[1:52]

Infinity / QNaN / SNaN / Zero

If WORD[1:8] =255 or WORD[1:31] =0
    frD[1:1] < WORD[0:1]
    frD[2] < WORD[1]
    frD[3] < WORD[1]
    frD[4] < WORD[1]
    frD[5:63] < WORD[2:31] || ^{29}0b0

For double-precision floating-point load instructions, no conversion is required as the data from memory is copied directly into the FPRs.

Many floating-point load instructions have an update form in which register **r**A is updated with the EA. For these forms, the effective address is placed into register **r**A and the memory element (word or double-word) addressed by the EA is loaded into the floating-point register specified by operand **fr**D.

4.5.8.2 Floating-Point Load Single Operands

If the operand falls in the range of a single denormalized number, the floating-point assist exception handler is invoked. Refer to **6.11.10 Floating-Point Assist Exception (0x00E00)** for additional information.

4.5.9 Floating-Point Store Instructions

There are two basic forms of the floating-point store instruction: single- and double-

precision. Because the FPRs support only floating-point, double-precision format, single-precision floating-point store instructions convert double-precision data to single-precision format before storing the operands. The conversion steps are described in **4.5.9.1 Double-Precision Conversion for Floating-Point Store Instructions**. **Table 4-19** is a summary of the floating-point store instructions.

Table 4-19 Floating-Point Store Instructions

Name	Mnemonic	Operand Syntax	Operation
Store Floating-Point Single-Precision	**stfs**	**fr**S,d(**r**A)	The EA is the sum (**r**A|0)+d. The contents of register **fr**S is converted to single-precision and stored into the word in memory addressed by the EA.
Store Floating-Point Single-Precision Indexed	**stfsx**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A|0)+(**r**B). The contents of register **fr**S is converted to single-precision and stored into the word in memory addressed by the EA.
Store Floating-Point Single-Precision with Update	**stfsu**	**fr**S,d(**r**A)	The EA is the sum (**r**A|0)+d. The contents of register **fr**S is converted to single-precision and stored into the word in memory addressed by the EA. The EA is placed into the register specified by operand **r**A.
Store Floating-Point Single-Precision with Update Indexed	**stfsux**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A|0)+(**r**B). The contents of register **fr**S is converted to single-precision and stored into the word in memory addressed by the EA. The EA is placed into the register specified by operand **r**A.
Store Floating-Point Double-Precision	**stfd**	**fr**S,d(**r**A)	The effective address is the sum (**r**A|0)+d. The contents of register **fr**S is stored into the double-word in memory addressed by the EA.
Store Floating-Point Double-Precision Indexed	**stfdx**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A|0)+(**r**B). The contents of register **fr**S is stored into the double-word in memory addressed by the EA.
Store Floating-Point Double-Precision with Update	**stfdu**	**fr**S,d(**r**A)	The effective address is the sum (**r**A|0)+d. The contents of register **fr**S is stored into the double-word in memory addressed by the EA. The EA is placed into register **r**A.

Table 4-19 Floating-Point Store Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Store Floating-Point Double-Precision with Update Indexed	**stfdux**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A|0)+(**r**B). The contents of register **fr**S is stored into the double-word in memory addressed by EA. The EA is placed into register **r**A.
Store Floating-Point as Integer Word	**stfiwx**	**fr**S,**r**A,**r**B	The 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.

4.5.9.1 Double-Precision Conversion for Floating-Point Store Instructions

The steps for converting single- to double-precision for floating-point store instructions are as follows:

Let WORD[0:31] be the word written in memory.

No Denormalization Required

```
If frS[1:11] > 896 or frS[1:63] = 0
    WORD[0:1] < frS[0:1]
    WORD[2:31]< frS[5:34]
```

Denormalization Required

```
If 874 ð frS[1:11] ð 896
    sign < frS[0]
    exp < frS[1:11] - 1023
    frac < 0b1 || frS[12:63]
    Denormalize operand
        Do while exp < -126
            frac <0 b0 || frac[0:62]
            exp < exp + 1
        End
    WORD0 < sign
    WORD[1:8] < 0x00
    WORD[9:31] < frac[1:23]
```

For double-precision floating-point store instructions, no conversion is required as the data from the FPRs is copied directly into memory. Many floating-point store instructions have an update form, in which register **r**A is updated with the effective address. For these forms, if operand **r**A ¦ 0, the effective address is placed into register **r**A.

Floating-point store instructions are listed in **Table 4-19**. Recall that **r**A, **r**B, and **r**D denote GPRs, while **fr**A, **fr**B, **fr**C, **fr**S and **fr**D denote FPRs.

4.5.9.2 Floating-Point Store-Single Operands

If the operand falls in the range of a single denormalized number, the floating-point assist exception handler is invoked.

If the operand is zero, it is converted to the correct signed zero in single-precision format.

If the operand is between the range of single denormalized and double denormalized, it is considered a programming error. The hardware handles this case as if the operand were single denormalized.

If the operand falls in the range of double denormalized numbers, it is considered a programming error. The hardware handles this case as if the operand were zero.

The following check is done on the stored operand in order to determine whether it is a denormalized single-precision operand and invoke the floating-point assist exception handler:

(FRS[1:11]) ≠ 0 AND FRS[1:11] ð 896

Refer to **6.11.10 Floating-Point Assist Exception (0x00E00)** for a complete description of handling denormalized floating-point numbers.

4.5.10 Floating-Point Move Instructions

Floating-point move instructions copy data from one floating-point register to another with data modifications as described for each instruction. These instructions do not modify the FPSCR. The condition register update option in these instructions controls the placing of result status into condition register field CR1. If the condition register update option is enabled, then CR1 is set, otherwise CR1 is unchanged. Floating-point move instructions are listed in **Table 4-20**.

Table 4-20 Floating-Point Move Instructions

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Move Register	**fmr** **fmr.**	**fr**D,**fr**B	The contents of register **fr**B is placed into **fr**D. **fmr** Floating-Point Move Register **fmr.** Floating-Point Move Register with CR Update. The dot suffix enables the update of the condition register.
Floating-Point Negate	**fneg** **fneg.**	**fr**D,**fr**B	The contents of register **fr**B with bit 0 inverted is placed into register **fr**D. **fneg** Floating-Point Negate **fneg.** Floating-Point Negate with CR Update. The dot suffix enables the update of the condition register.
Floating-Point Absolute Value	**fabs** **fabs.**	**fr**D,**fr**B	The contents of **fr**B with bit 0 cleared to zero is placed into **fr**D. **fabs** Floating-Point Absolute Value **fabs.** Floating-Point Absolute Value with CR Update. The dot suffix enables the update of the condition register.
Floating-Point Negative Absolute Value	**fnabs** **fnabs.**	**fr**D,**fr**B	The contents of **fr**B with bit 0 set to one is placed into **fr**D. **fnabs** Floating-Point Negative Absolute Value **fnabs.** Floating-Point Negative Absolute Value with CR Update. The dot suffix enables the update of the condition register.

4.6 Flow Control Instructions

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

If an interlock is detected, the branch is considered unresolved and the direction of the branch is predicted using static branch prediction as described in **Table 4-21**. The interlock is monitored while instructions are fetched for the predicted branch. When the interlock is cleared, the branch processor determines whether the prediction was correct based on the value of the condition register bit. If the prediction is correct, the branch is considered completed and instruction fetching continues. If the prediction is incorrect, the prefetched instructions are purged, and instruction fetching continues along the alternate path.

When the branch instructions contain immediate addressing operands, the target addresses can be computed sufficiently ahead of the branch instruction that instructions can be prefetched along the target path. If the branch instructions use the link and count registers, instructions along the target path can be prefetched if the link or count register is loaded sufficiently ahead of the branch instruction.

Branching can be conditional or unconditional, and the return address can optionally be provided. If the return address is to be provided, the effective address of the instruction following the branch instruction is placed in the link register after the branch target address has been computed. This is done regardless of whether the branch is taken.

4.6.1 Branch Instruction Address Calculation

Branch instructions can change the sequence of instruction execution. Instruction addresses are always assumed to be on word boundaries; therefore the processor ignores the two low-order bits of the generated branch target address.

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

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

4.6.1.1 Branch Relative Address Mode

Instructions that use branch relative addressing generate the next instruction address by sign extending and appending 0b00 to the immediate displacement operand (LI) and adding the resultant value to the current instruction address. Branches using this address mode have the absolute addressing option disabled (AA, bit 30 in the instruction encoding, equals zero). If the link register update option is enabled (LK, bit 31 in the instruction encoding, equals one), the effective address of the instruction following the branch instruction is placed in the link register.

Figure 4-6 shows how the branch target address is generated when using the branch relative addressing mode.

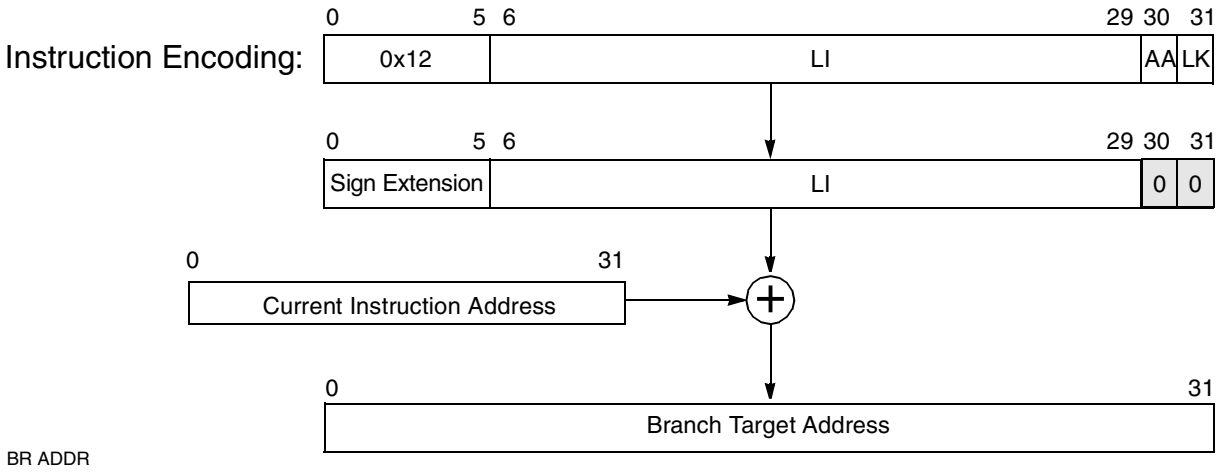

Figure 4-6 Branch Relative Addressing

4.6.1.2 Branch Conditional Relative Address Mode

If the branch conditions are met, instructions that use the branch conditional relative address mode generate the next instruction address by sign extending and appending 0b00 to the immediate displacement operand (BD) and adding the resultant value to the current instruction address. Branches using this address mode have the absolute addressing option disabled (AA, bit 30 in the instruction encoding, equals zero). If the link register update option is enabled (LK, bit 31 in the instruction encoding, equals one), the effective address of the instruction following the branch instruction is placed in the link register.

Figure 4-7 shows how the branch target address is generated when using the branch conditional relative addressing mode.

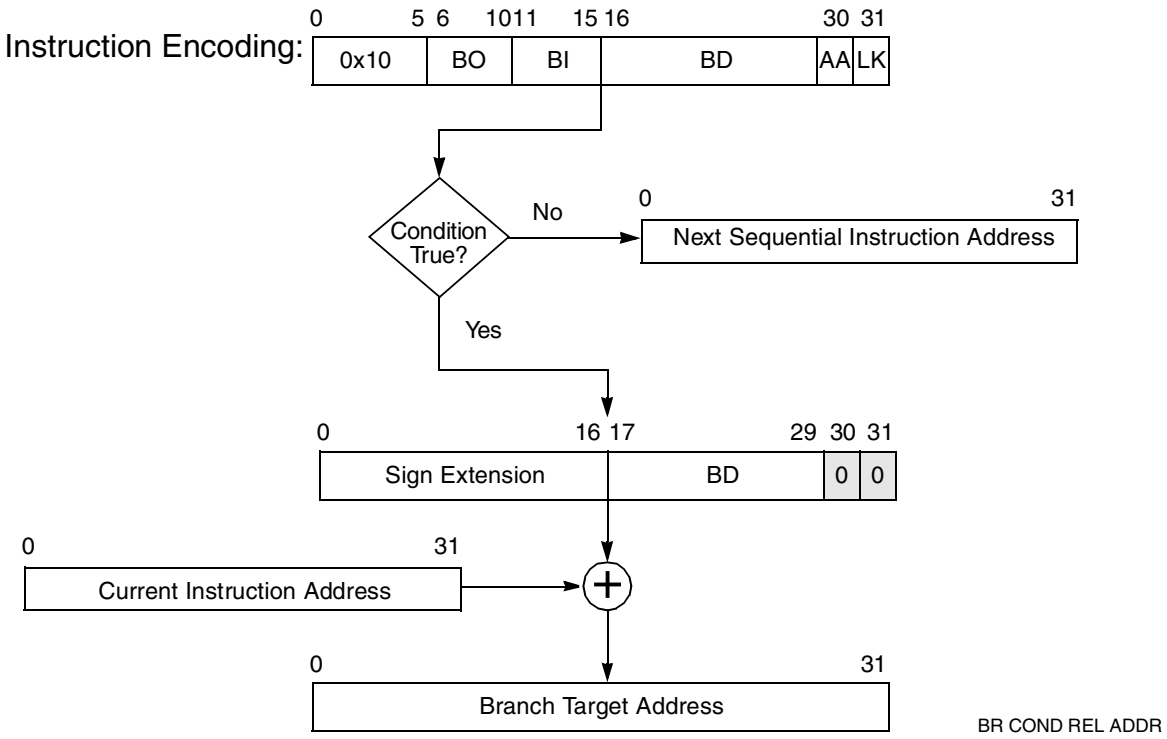

Figure 4-7 Branch Conditional Relative Addressing

4.6.1.3 Branch to Absolute Address Mode

Instructions that use branch to absolute address mode generate the next instruction address by sign extending and appending 0b00 to the LI operand. Branches using this address mode have the absolute addressing option enabled (AA, bit 30 in the instruction encoding, equals one). If the link register update option is enabled (LK, bit 31 in the instruction encoding, equals one), the effective address of the instruction following the branch instruction is placed in the link register.

Figure 4-8 shows how the branch target address is generated when using the branch to absolute address mode.

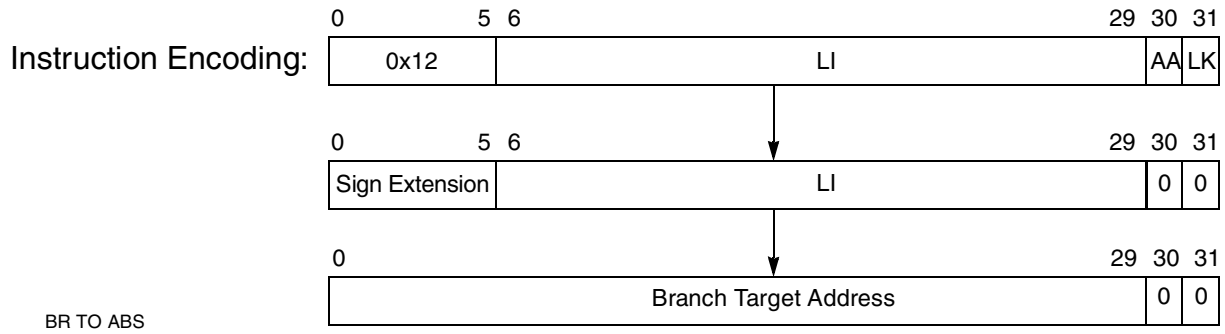

Figure 4-8 Branch to Absolute Addressing

4.6.1.4 Branch Conditional to Absolute Address Mode

If the branch conditions are met, instructions that use the branch conditional to absolute address mode generate the next instruction address by sign extending and appending 0b00 to the BD operand. Branches using this address mode have the absolute addressing option enabled (AA, bit 30 in the instruction encoding, equals one). If the link register update option is enabled (LK, bit 31 in the instruction encoding, equals one), the effective address of the instruction following the branch instruction is placed in the link register.

Figure 4-9 shows how the branch target address is generated when using the branch conditional to absolute address mode.

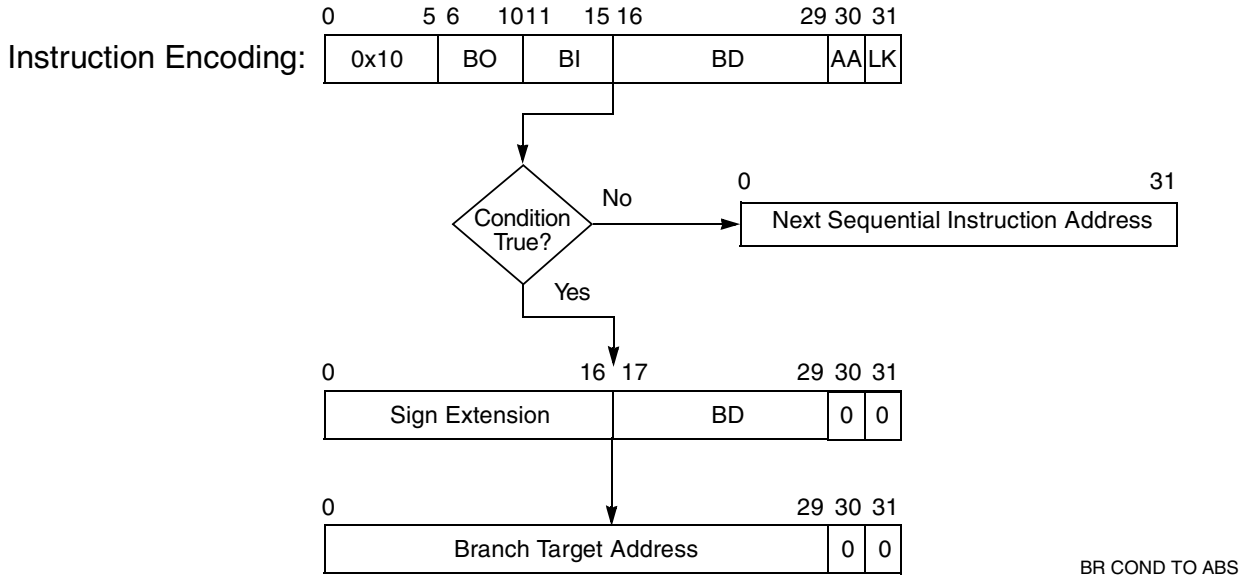

Figure 4-9 Branch Conditional to Absolute Addressing

4.6.1.5 Branch Conditional to Link Register Address Mode

If the branch conditions are met, the branch conditional to link register instruction generates the next instruction address by fetching the contents of the link register and clearing the two low order bits to zero. If the link register update option is enabled (LK, bit 31 in the instruction encoding, equals one), the effective address of the instruction following the branch instruction is placed in the link register.

Figure 4-10 shows how the branch target address is generated when using the branch conditional to link register address mode.

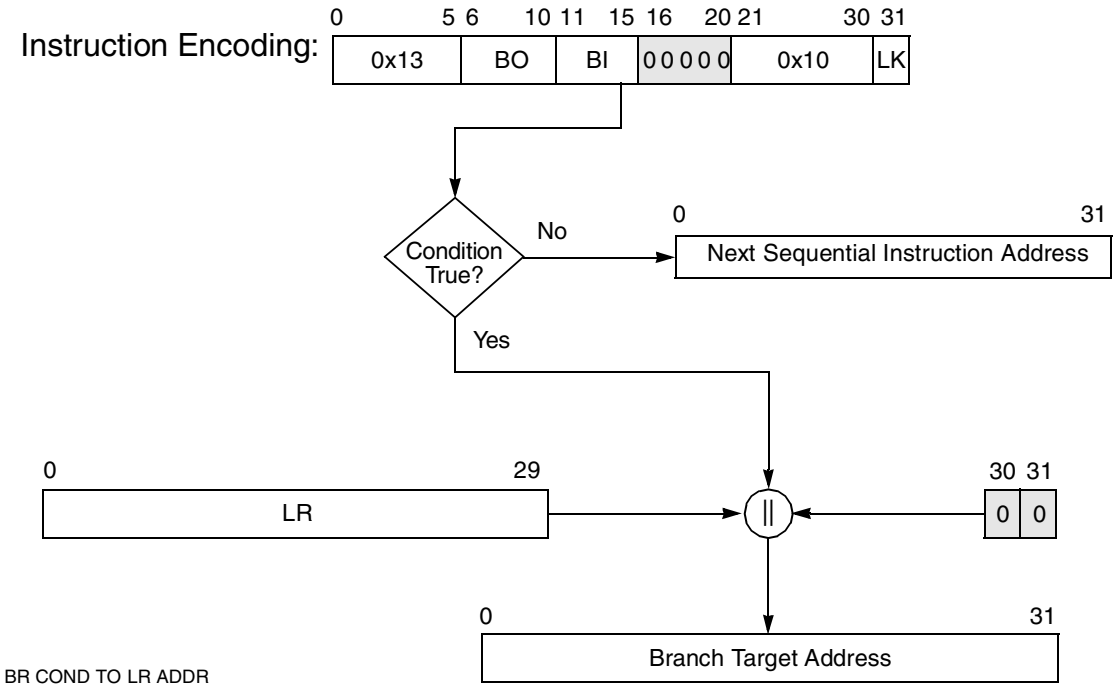

Figure 4-10 Branch Conditional to Link Register Addressing

4.6.1.6 Branch Conditional to Count Register

If the branch conditions are met, the branch conditional to count register instruction generates the next instruction address by fetching the contents of the count register and clearing the two low order bits to zero. If the link register update option is enabled (LK, bit 31 in the instruction encoding, equals one), the effective address of the instruction following the branch instruction is placed in the link register.

Figure 4-11 shows how the branch target address is generated when using the branch conditional to count register address mode.

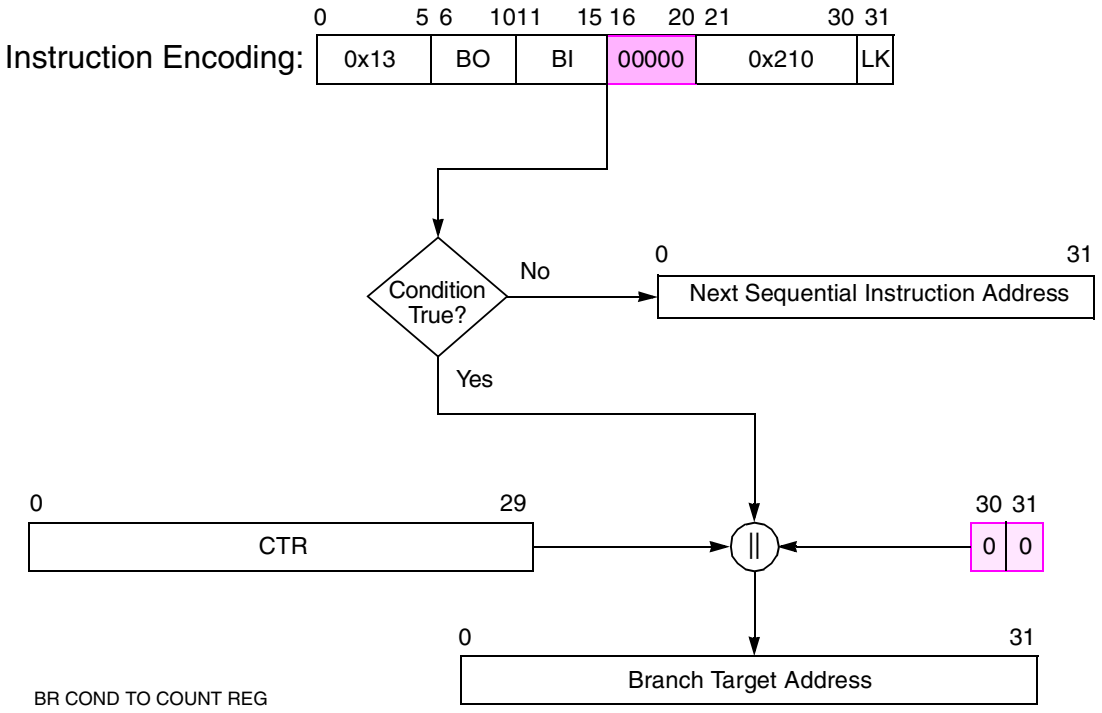

Figure 4-11 Branch Conditional to Count Register Addressing

4.6.2 Conditional Branch Control

For branch conditional instructions, the BO and BI operands specify the conditions under which the branch is taken.

4.6.2.1 BO Operand and Branch Prediction

The encodings for the BO operand are shown in **Table 4-21**.

Table 4-21 BO Operand Encodings

BO[1]	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.

NOTES:
1. The *z* indicates a bit that must be zero; otherwise, the instruction form is invalid. The *y* bit provides a hint about whether a conditional branch is likely to be taken.

The first four bits of the BO operand specify how the branch is affected by or affects the condition and count registers. The fifth bit, shown in **Table 4-21** as having the value *y*, is used for branch prediction. The branch always encoding of the BO operand does not have a *y* bit.

Clearing the *y* bit to zero indicates that the following behavior is likely:

- For **bc**x with a negative value in the displacement operand, the branch is taken.
- In all other cases (**bc**x with a non-negative value in the displacement operand, **bclr**x, or **bcctr**x), the branch is not taken.

Setting the *y* bit to one reverses the preceding indications.

Note that branch prediction occurs for branches to the LR or CTR only if the target address is ready.

The sign of the displacement operand is used as described above even if the target is an absolute address. The default value for the *y* bit should be zero, and should only be set to one if software has determined that the prediction corresponding to *y* = one is more likely to be correct than the prediction corresponding to *y* = zero. Software that does not compute branch predictions should set the *y* bit to zero.

For all three of the branch conditional instructions, the branch should be predicted to be taken if the value of the following expression is one, and to fall through if the value is zero.

((BO[0] & BO[2]) | S) ⊕ BO[4]

In the expression above, S (bit 16 of the branch conditional instruction coding) is

the sign bit of the displacement operand if the instruction has a displacement operand and is zero if the operand is reserved. BO[4] is the *y* bit, or zero for the branch always encoding of the BO operand. (Advantage is taken of the fact that, for **bclr**x and **bcctr**x, bit 16 of the instruction is part of a reserved operand and therefore must be zero.)

4.6.2.2 BI Operand

The 5-bit BI operand in branch conditional instructions specifies which of the 32 bits in the CR represents the condition to test.

4.6.2.3 Simplified Mnemonics for Conditional Branches

To provide a simplified mnemonic for every possible combination of BO and BI fields would require 2^{10} = 1024 mnemonics, most of which would be only marginally useful. The abbreviated set found in **E.5 Simplified Mnemonics for Branch Instructions** is intended to cover the most useful cases. Unusual cases can be coded using a basic branch conditional mnemonic (**bc**, **bclr**, **bcctr**) with the condition to be tested specified as a numeric operand.

4.6.3 Branch Instructions

Table 4-22 describes the RCPU branch instructions.

Table 4-22 Branch Instructions

Name	Mnemonic	Operand Syntax	Operation	
Branch	**b** **ba** **bl** **bla**	imm_addr	**b** **ba** **bl** **bla**	Branch. Branch to the address computed as the sum of the immediate address and the address of the current instruction. Branch Absolute. Branch to the absolute address specified. Branch then Link. Branch to the address computed as the sum of the immediate address and the address of the current instruction. The instruction address following this instruction is placed into the link register (LR). Branch Absolute then Link. Branch to the absolute address specified. The instruction address following this instruction is placed into the link register (LR).

Table 4-22 Branch Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Branch Conditional	**bc** **bca** **bcl** **bcla**	BO,BI, target_addr	The BI operand specifies the bit in the condition register (CR) to be used as the condition of the branch. The BO operand is used as described in Table 4-21. **bc** Branch Conditional. Branch conditionally to the address computed as the sum of the immediate address and the address of the current instruction. **bca** Branch Conditional Absolute. Branch conditionally to the absolute address specified. **bcl** Branch Conditional then Link. Branch conditionally to the address computed as the sum of the immediate address and the address of the current instruction. The instruction address following this instruction is placed into the link register. **bcla** Branch Conditional Absolute then Link. Branch conditionally to the absolute address specified. The instruction address following this instruction is placed into the link register.
Branch Conditional to Link Register	b**clr** **bclrl**	BO,BI	The BI operand specifies the bit in the condition register to be used as the condition of the branch. The BO operand is used as described in Table 5–21. **bclr** Branch Conditional to Link Register. Branch conditionally to the address in the link register. **bclrl** Branch Conditional to Link Register then Link. Branch conditionally to the address specified in the link register. The instruction address following this instruction is then placed into the link register.
Branch Conditional to Count Register	**bcctr** **bcctrl**	BO,BI	The BI operand specifies the bit in the condition register to be used as the condition of the branch. The BO operand is used as described in Table 5–21. **bcctr** Branch Conditional to Count Register. Branch conditionally to the address specified in the count register. **bcctrl** Branch Conditional to Count Register then Link. Branch conditionally to the address specified in the count register. The instruction address following this instruction is placed into the link register. **Note:** If the “decrement and test CTR” option is specified (BO[2]=0), the instruction form is invalid.

4.6.4 Condition Register Logical Instructions

Similar to the system call (**sc**) instruction, condition register logical instructions, shown in **Table 4-23**, and the move condition register field (**mcrf**) instruction are defined as flow control instructions, although they are executed by the IU.

Note that if the link register update option (LR) is enabled for any of these instructions, the PowerPC architecture defines these forms of the instructions as invalid.

Table 4-23 Condition Register Logical Instructions

Name	Mnemonic	Operand Syntax	Operation
Condition Register AND	**crand**	**crb**D,**crb**A,**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.
Condition Register OR	**cror**	**crb**D,**crb**A,**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.
Condition Register XOR	**crxor**	**crb**D,**crb**A,**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. The result is placed into the condition register bit specified by **crb**D.
Condition Register NAND	**crnand**	**crb**D,**crb**A,**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 complemented result is placed into the condition register bit specified by **crb**D.
Condition Register NOR	**crnor**	**crb**D,**crb**A,**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 complemented result is placed into the condition register bit specified by **crb**D.
Condition Register Equivalent	**creqv**	**crb**D,**crb**A, **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. The complemented result is placed into the condition register bit specified by **crb**D.
Condition Register AND with Complement	**crandc**	**crb**D,**crb**A, **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.
Condition Register OR with Complement	**crorc**	**crb**D,**crb**A, **crb**B	The bit in the condition register specified by **crb**A is ORed 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.
Move Condition Register Field	**mcrf**	**crf**D,**crf**S	The contents of **crf**S are copied into **crf**D. No other condition register fields are changed.

Refer to **E.6 Simplified Mnemonics for Condition Register Logical Instructions** for simplified mnemonics.

4.6.5 System Linkage Instructions

This section describes the system linkage instructions (see **Table 4-29**). The system call (**sc**) instruction permits a program to call on the system to perform a service and the system to return from performing a service or from processing an exception.

Table 4-24 System Linkage Instructio ns

Name	Mnemonic	Operand Syntax	Operand Syntax
System Call	**sc**	—	When executed, the effective address of the instruction following the **sc** instruction is placed into SRR0. MSR[16:31] are placed into SRR1[16:31], and SRR1[0:15] are set to undefined values. Then a system call exception is generated. The exception causes the next instruction to be fetched from offset 0xC00 from the base physical address indicated by the new setting of MSR[IP]. Refer to **6.11.8 System Call Exception (0x00C00)** for more information. This instruction is context synchronizing.
Return from Interrupt	**rfi**	—	SRR1[16:31] are placed into MSR[16:31], then the next instruction is fetched, under control of the new MSR value, from the address SRR0[0:29] || 0b00. This is a supervisor-level, context-synchronizing instruction.

4.6.6 Simplified Mnemonics for Branch and Flow Control Instructions

To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for the most frequently used forms of branch conditional, trap, and certain other instructions; for more information, see **APPENDIX E SIMPLIFIED MNEMONICS**.

Mnemonics are provided so that branch conditional instructions can be coded with the condition as part of the instruction mnemonic rather than as a numeric operand. Some of these are shown as examples with the branch instructions.

PowerPC-compliant assemblers provide the mnemonics and symbols listed here and possibly others.

4.6.7 Trap Instructions

The trap instructions shown in **Table 4-25** are provided to test for a specified set of conditions. If any of the conditions tested by a trap instruction are met, the system trap handler is invoked. If the tested conditions are not met, instruction execution continues normally.

Table 4-25 Trap Instructions

Name	Mnemonic	Operand Syntax	Operand Syntax
Trap Word Immediate	**twi**	TO,**r**A,SIMM	The contents of **r**A is compared with the sign-extended SIMM operand. If any bit in the TO operand is set to one and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.
Trap Word	**tw**	TO,**r**A,**r**B	The contents of **r**A is compared with the contents of **r**B. If any bit in the TO operand is set to one and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

The trap instructions evaluate a trap condition as follows:

The contents of register **r**A is compared with either the sign-extended SIMM field or with the contents of register **r**B, depending on the trap instruction. The comparison results in five conditions which are ANDed with operand TO. If the result is not zero, the trap exception handler is invoked. These conditions are provided in **Table 4-26**.

Table 4-26 TO Operand Bit Encoding

TO Bit	ANDed with Condition
0	Less than
1	Greater than
2	Equal
3	Logically less than
4	Logically greater than

A standard set of codes has been adopted for the most common combinations of trap conditions. Refer to **E.7 Simplified Mnemonics for Trap Instructions** for a description of these codes and of simplified mnemonics employing them.

4.7 Processor Control Instructions

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

4.7.1 Move to/from Machine State Register and Condition Register Instructions

Table 4-27 summarizes the instructions for reading from or writing to the machine state register and the condition register.

Table 4-27 Move to/from Machine State Register/Condition Register Instructions

Name	Mnemonic	Operand Syntax	Operation
Move to Condition Register Fields	**mtcrf**	CRM,**r**S	The contents of **r**S are placed into the condition register under control of the field mask specified by operand CRM. The field mask identifies the 4-bit fields affected. Let *i* be an integer in the range 0-7. If CRM(*i*) = 1, then CR field *i* (CR bits 4**i* through 4**i*+3) is set to the contents of the corresponding field of **r**S.
Move to Condition Register from XER	**mcrxr**	**crf**D	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 to zero.
Move from Condition Register	**mfcr**	**r**D	The contents of the condition register are placed into **r**D.
Move to Machine State Register	**mtmsr**	**r**S	The contents of **r**S are placed into the MSR. This instruction is a supervisor-level instruction and is context synchronizing.
Move from Machine State Register	**mfmsr**	**r**D	The contents of the MSR are placed into **r**D. This is a supervisor-level instruction.

4.7.2 Move to/from Special Purpose Register Instructions

Simplified mnemonics are provided for the **mtspr** and **mfspr** instructions so they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the two instructions. (See **Table 4-28**.) Refer to **E.8 Simplified Mnemonics for Special-Purpose Registers** for a complete list of these mnemonics.

Table 4-28 Move to/from Special Purpose Register Instructions

Name	Mnemonic	Operand Syntax	Operation
Move to Special Purpose Register	**mtspr**	SPR,**r**S	The SPR field denotes a special purpose register, encoded as shown in **Table 4-29** and **Table 4-30** below. The contents of **r**S are placed into the designated SPR. Simplified mnemonic examples: **mtxer r**A **mtspr** 1,**r**A **mtlr r**A **mtspr** 8,**r**A **mtctr r**A **mtspr** 9,**r**A
Move from Special Purpose Register	**mfspr**	**r**D,SPR	The SPR field denotes a special purpose register, encoded as shown in **Table 4-29** and **Table 4-30** below. The contents of the designated SPR are placed into **r**D. Simplified mnemonic examples: **mfxer r**A **mfspr r**A,1 **mflr r**A **mfspr r**A,8 **mfctr r**A **mfspr r**A,9

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].

Table 4-29 summarizes SPR encodings to which the RCPU permits user-level access.

Table 4-29 User-Level SPR Encodings

Decimal Value in rD	SPR[0:4] SPR[5:9]	Register Name	Description
1	0b00001 00000	XER	Integer exception register
8	0b01000 00000	LR	Link register
9	0b01001 00000	CTR	Count register
268	0b01100 01000	TBL	Time base — lower (read only)
269	0b01101 01000	TBU	Time base — upper (read only)

Table 4-30 summarizes SPR encodings that the RCPU permits at the supervisor level.

Table 4-30 Supervisor-Level SPR Encodings

Decimal Value in rD[1]	SPR[0:4] SPR[5:9]	Register Name	Description
18	0b10010 00000	DSISR	DAE/source instruction service register
19	0b10011 00000	DAR	Data address register
22	0b10110 00000	DEC	Decrementer register
26	0b11010 00000	SRR0	Save and restore register 0
27	0b11011 00000	SRR1	Save and restore register 1
80	0b10000 00010	EIE	External interrupt enable (write only)
81	0b10001 00010	EID	External interrupt disable (write only)
82	0b10010 00010	NRI	Non-recoverable exception
272	0b10000 01000	SPRG0	SPR general 0
273	0b10001 01000	SPRG1	SPR general 1
274	0b10010 01000	SPRG2	SPR general 2
275	0b10011 01000	SPRG3	SPR general 3
284	0b11100 01000	TBL[2]	Time base — lower (write only)
285	0b11101 01000	TBU[2]	Time base — upper (write only)
287	0b11111 01000	PVR	Processor version register (read only)
560	0b10000 10001	ICCST	I-Cache Control and Status Register
561	0b10001 10001	ICADR	I-cache address register
562	0b10010 10001	ICDAT	I-cache data port
1022	0b11110 11111	FPECR	Floating-point exception cause register

NOTES:

1. If the SPR field contains any value other than one of the values shown in **Table 4-30**, the instruction form is invalid. For an invalid instruction form in which SPR[0]=1, either a privileged instruction type program exception or software emulation exception is generated if the instruction is executed by a user-level program. (Refer to the discussion of these two exception types in **SECTION 6 EXCEPTIONS** for more information.) If the instruction is executed by a supervisor-level program, the software emulation exception handler is invoked.
   SPR[0] = 1 if and only if writing the register is supervisor-level. Execution of this instruction specifying a defined and supervisor-level register when MSR[PR] = 1 results in a privileged instruction type program exception.
2. The PowerPC architecture defines the encodings as TBRs, although it is the same as the SPR encodings. Moving to the time base is performed by the **mtspr** instruction, and moving from the time base is performed by the **mftb** instruction.

Table 4-31 summarizes SPR encodings that the RCPU permits in debug mode, or in supervisor mode when debug mode is not enabled out of reset.

Table 4-31 Development Support SPR Encodings

Decimal Value in rD	SPR[0:4] SPR[5:9]	Register Name	Description
144	0b10000 00010	CMPA	Comparator A Value Register
145	0b10001 00010	CMPB	Comparator B Value Register
146	0b10010 00010	CMPC	Comparator C Value Register
147	0b10011 00010	CMPD	Comparator D Value Register
148	0b10100 00010	ECR	Exception Cause Register
149	0b10101 00010	DER	Debug Enable Register
150	0b10110 00010	COUNTA	Breakpoint Counter A Value and Control
151	0b10111 00010	COUNTB	Breakpoint Counter B Value and Control
152	0b11000 00010	CMPE	Comparator E Value Register
153	0b11001 00010	CMPF	Comparator F Value Register
154	0b11010 00010	CMPG	Comparator G Value Register
155	0b11011 00010	CMPH	Comparator H Value Register
156	0b11100 00010	LCTRL1	L-Bus Support Comparators Control 1
157	0b11101 00010	LCTRL2	L-Bus Support Comparators Control 2
158	0b11110 00010	ICTRL	I-Bus Support Control
159	0b11111 00010	BAR	Breakpoint Address Register
630	0b10110 10011	DPDR	Development Port Data Register

4.7.3 Move from Time Base Instruction

The **mftb** instruction is used to read from the time base register. The instruction is permitted at the user or supervisor privilege level.

Simplified mnemonics for the **mftb** instruction allow it to be coded with the TBR name as part of the mnemonic. Refer to **E.8 Simplified Mnemonics for Special-Purpose Registers** for details. Notice that the simplified mnemonics for move from time base and move from time base upper 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.

Table 4-32 Move from Time Base Instruction

Name	Mnemonic	Operand Syntax	Operation
Move from Time Base	**mftb**	**r**D,TBR	The TBR field denotes either the time base lower (TBL) or time base upper (TBU), encoded as shown in **Table 4-33**. The contents of the designated register are copied to **r**D.

Table 4-33 summarizes the time base (TBL/TBU) register encodings to which user-level access read access (using the **mftb** instruction) is permitted.

Table 4-33 User-Level TBR Encodings

Decimal Value in rD	SPR[0:4] SPR[5:9]	Register Name	Description
268	0b01100 01000	TBL	Time base lower (read only)
269	0b01101 01000	TBU	Time base upper (read only)

Writing to the time base is permitted at the supervisor privilege level only and is accomplished with the **mtspr** instruction (see **4.7.2 Move to/from Special Purpose Register Instructions**) or the **mttb** simplified mnemonic (see **E.8 Simplified Mnemonics for Special-Purpose Registers**).

4.8 Memory Synchronization Instructions

Memory synchronization instructions can 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 and by other mechanisms that access memory.

The synchronize (**sync**) instruction delays execution of subsequent instructions until all previous instructions have completed (i.e., all internal pipeline stages and instruction buffers have emptied), all previous memory accesses are performed globally, and the **sync** or **eieio** operation is broadcast onto the external bus interface. This set of conditions is referred to as execution serialization (or simply serialization).

The enforce in-order execution of I/O (**eieio**) instruction serializes load/store instructions. No load or store instruction following **eieio** is issued until all loads and stores preceding **eieio** have completed execution.

The instruction synchronize (**isync**) instruction causes the RCPU to halt instruction fetch until all instructions currently in the processor have completed execution, i.e., all issued instructions as well as the pre-fetched instructions waiting to be issued. This condition is referred to as fetch serialization.

The proper use of the load word and reserve indexed (**lwarx**) and store word conditional indexed (**stwcx.**) instructions allows programmers to emulate common semaphore operations such as “test and set”, “compare and swap”, “exchange memory”, and “fetch and add”. Examples of these semaphore operations can be found in **APPENDIX D SYNCHRONIZATION PROGRAMMING EXAMPLES**. The **lwarx** instruction must be paired with an **stwcx.** instruction with the same effective address used for both instructions of the pair. The reservation granularity is 32 bytes.

The concept behind the use of the **lwarx** and **stwcx.** instructions is that a processor may load a semaphore from memory, compute a result based on the value of the semaphore, and conditionally store it back to the same location. The conditional store is performed based on the existence of a reservation established by the preceding **lwarx**. If the reservation exists when the store is executed, the store is performed and a bit is set to one in the condition register. If the reservation does not exist when the store is executed, the target memory location is not modified and a bit is set to zero in the condition register.

The **lwarx** and **stwcx.** primitives allow software to read a semaphore, compute a result based on the value of the semaphore, store the new value back into the semaphore location only if that location has not been modified since it was first read, and determine if the store was successful. If the store was successful, the sequence of instructions from the read of the semaphore to the store that updated the semaphore appear to have been executed atomically (that is, no other processor or mechanism modified the semaphore location between the read and the update), thus providing the equivalent of a real atomic operation. However, other processors may have read from the location during this operation.

The **lwarx** and **stwcx.** instructions require the EA to be aligned. Exception handling software should not attempt to emulate a misaligned **lwarx** or **stwcx.** instruction, because there is no correct way to define the address associated with the reservation.

In general, the **lwarx** and **stwcx.** instructions should be used only in system programs, which can be invoked by application programs as needed.

At most one reservation exists at a time on a given processor. The address associated with the reservation can be changed by a subsequent **lwarx** instruction. The conditional store is performed based on the existence of a reservation established by the preceding **lwarx** regardless of whether the address generated by the **lwarx** matches that generated by the **stwcx.** A reservation held by the processor is cleared by any of the following:

- execution of an **stwcx.** instruction to any address
- execution of an **sc** instruction
- execution of an instruction that causes an exception
- occurrence of an asynchronous exception
- attempt by some other device to modify a location in the reservation granularity (32 bytes)

When an **lwarx** instruction is executed, the load/store unit issues a cycle to the load/store bus with a special attribute.

In case of an external memory access, this attribute causes the external bus interface (EBI) to set a storage reservation on the cycle address. The EBI must either snoop the external bus or receive some indication from external snoop logic in case the storage reservation is broken by some other processor accessing the same location. When an **stwcx.** instruction to external memory is executed, the EBI checks if the reservation was lost. If so, the cycle is blocked from going to the external bus, and the EBI notifies the LSU that the **stwcx.** instruction did not complete.

The RCPU memory synchronization instructions are summarized in **Table 4-34**.

Table 4-34 Memory Synchronization Instructions

Name	Mnemonic	Operand Syntax	Operation
Enforce In-Order Execution of I/O	**eieio**	—	The **eieio** instruction provides an ordering function for the effects of load and store instructions executed by a given processor. Executing an **eieio** instruction ensures that all memory accesses previously initiated by the given processor are complete with respect to main memory before allowing any memory accesses subsequently initiated by the given processor to access main memory.
Instruction Synchronize	**isync**	—	This instruction causes instruction fetch to be halted until all instructions currently in the processor have completed execution, i.e., all issued instructions as well as the pre-fetched instructions waiting to be issued. This instruction has no effect on other processors or on their caches.
Load Word and Reserve Indexed	**lwarx**	**r**D,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). The word in memory addressed by the EA is loaded into register **r**D. This instruction creates a reservation for use by an **stwcx.** instruction. An address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation. The EA must be a multiple of four. If it is not, the alignment exception handler is invoked.
Store Word Conditional Indexed	**stwcx**.	**r**S,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). If a reservation exists, register **r**S is stored into the word in memory addressed by the EA and the reservation is cleared. If a reservation does not exist, the instruction completes without altering memory. The EQ bit in the condition register field CR0 is modified to reflect whether the store operation was performed (i.e., whether a reservation existed when the **stwcx.** instruction began execution). If the store was completed successfully, the EQ bit is set to one. The EA must be a multiple of four; otherwise, the alignment exception handler is invoked.

Table 4-34 Memory Synchronization Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Synchronize	**sync**	—	Executing a **sync** instruction ensures that all instructions previously initiated by the given processor appear to have completed before any subsequent instructions are initiated by the given processor. When the **sync** instruction completes, all memory accesses initiated by the given processor prior to the **sync** will have been performed with respect to all other mechanisms that access memory. The **sync** instruction can be used to ensure that the results of all stores into a data structure, performed in a critical section of a program, are seen by other processors before the data structure is seen as unlocked.

4.9 Memory Control Instructions

This section describes memory control instructions. In the RCPU, only one such instruction is supported: Instruction cache block invalidate (**icbi**).

Table 4-35 Instruction Cache Management Instruction

Name	Mnemonic	Operand Syntax	Operation
Instruction Cache Block Invalidate	**icbi**	**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). This instruction causes any subsequent fetch request for an instruction in the block to not find the block in the cache and to be sent to storage. The instruction causes the target block in the instruction cache of the executing processor to be marked invalid. If the target block is not accessible to the program for loads, the system data storage error handler may be invoked. This is a supervisor-level instruction.

4.10 Recommended Simplified Mnemonics

To simplify assembly language programs, a set of simplified mnemonics is provided for some of the most frequently used instructions such as no-op, load immediate, load address, move register, and complement register). PowerPC compliant assemblers provide the simplified mnemonics listed in **E.9 Recommended Simplified Mnemonics**. Programs written to be portable across the various assemblers for the PowerPC architecture should not assume the existence of mnemonics not defined in this manual.

For a complete list of simplified mnemonics, see **APPENDIX E SIMPLIFIED MNEMONICS**.