



from

WRITE GREAT CODE

Volume 2: Thinking Low-Level, Writing High-Level

ONLINE APPENDIX B

The Minimal PowerPC Instruction Set

by Randall Hyde



**NO STARCH
PRESS**

San Francisco

WRITE GREAT CODE, Volume 2. Copyright © 2006 by Randall Hyde. ISBN 1-59327-065-8.

All Rights Reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

For information on book distributors or translations, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.

555 De Haro Street, Suite 250, San Francisco, CA 94107

phone: 415.863.9900; fax: 415.863.9950; info@nostarch.com; www.nostarch.com

The information in this online appendix is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this material, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

Library of Congress Cataloging-in-Publication Data (Volume 1)

Hyde, Randall.

Write great code : understanding the machine / Randall Hyde.

p. cm.

ISBN 1-59327-003-8

1. Computer programming. 2. Computer architecture. I. Title.

QA76.6.H94 2004

005.1--dc22

2003017502

If you haven't purchased a copy of *Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level* and would like to do so, please go to www.nostarch.com.

B

THE MINIMAL POWERPC INSTRUCTION SET



Although the PowerPC CPU family supports hundreds of instructions, few compilers actually use all of these instructions. If you're wondering why compilers don't use more of the available instructions, the answer is because many of them have become obsolete over time as newer instructions have reduced the need for older instructions. Some instructions, such as the PowerPC's *AltaVec* instructions, simply do not correspond to operations you'd normally perform in an HLL. Therefore, compilers rarely generate these types of machine instructions (such instructions generally appear only in handwritten assembly language programs). Therefore, you don't need to learn the entire PowerPC instruction set to study compiler output. Instead, you need only learn the handful of instructions that compilers actually emit on the PowerPC. That's the purpose of this appendix, to describe those few instructions that compilers actually use.

Many PowerPC instructions take multiple forms depending on whether they modify the condition-code and XER registers. An unadorned instruction mnemonic does not modify either register. A dot suffix (.) on certain

instructions tells the CPU to update the condition-code CR0 bits based on the result of the operation. An *o* suffix tells the CPU to update the overflow and summary overflow bits in the XER register. Finally, an *o.* suffix tells the CPU to update the bits in CR0 and the XER register. The following descriptions group instructions together that differ only by these suffixes.

B.1 add, add., addo, addo.

The `add` instruction requires three register operands—a destination register and two source registers. This instruction computes the sum of the values in the two source registers and stores the sum into the destination register.

Table B-1: Gas Syntax for `add`

| Instruction | Description |
|---------------------------------|---|
| <code>add Rd, Rs1, Rs2</code> | $Rd := Rs1 + Rs2$ <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| <code>add. Rd, Rs1, Rs2</code> | $Rd := Rs1 + Rs2$ CRO reflects the result of the sum. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| <code>addo Rd, Rs1, Rs2</code> | $Rd := Rs1 + Rs2$ The overflow and summary overflow bits in XER are set if a signed overflow occurs. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| <code>addo. Rd, Rs1, Rs2</code> | $Rd := Rs1 + Rs2$ CRO reflects the result of the sum. The overflow and summary overflow bits in XER are set if a signed overflow occurs. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |

Table B-2: CRO Settings for `add.` and `addo.`

| Flag | Setting |
|------|--|
| LT | Set if the sum (signed) is less than zero. |
| GT | Set if the sum (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | The summary overflow bit from the XER is copied to this field after computing the sum. |

Table B-3: XER Settings for `addo` and `addo.`

| Flag | Setting |
|------|---|
| OV | Set if a signed overflow occurred during the execution of the instruction. |
| SO | Set if the SO bit was previously set, or if a signed overflow occurred during the execution of the instruction. |
| CA | Unaffected |

B.2 addi

The `addi` instruction (add immediate) adds a constant to the contents of a source register and stores the sum into a destination register. The constant is limited to a signed 16-bit value (which the instruction sign extends to 32 bits prior to use). This instruction does not affect any flags or the overflow bit.

The `addi` instruction treats `R0` differently than the other registers. If you specify `R0` as the source register, the `addi` instruction uses the value zero rather than the value held in the `R0` register. In this case, the `addi` instruction acts as a “load immediate with sign extension” instruction (because adding an immediate constant with zero simply produces that constant). Though the PowerPC doesn’t have an actual “load immediate” instruction, most assemblers assemble the `li` instruction into the `addi` opcode.

You will also discover that there is no “subtract immediate” instruction, even though assemblers like Gas support that mnemonic. Gas (and other PowerPC assemblers) compile a `subi` instruction into an `addi` instruction after negating the immediate operand.

Table B-4: Gas Syntax for `addi`

| Instruction | Description |
|-------------------------------------|---|
| <code>addi Rd, Rs1, constant</code> | $Rd := Rs1 + constant$ <i>d</i> and <i>s1</i> are register numbers in the range 0..31. |

B.3 addis

The `addis` instruction (add immediate, shifted) shifts a 16-bit constant to the left 16 bits, adds this to the value from a source register, and then stores the sum into a destination register. This instruction does not affect any flags or the overflow bit.

The `addis` instruction treats `R0` differently than the other registers. If you specify `R0` as the source register, the `addis` instruction uses the value zero rather than the value held in the `R0` register.

Table B-5: Gas Syntax for `addis`

| Instruction | Description |
|--------------------------------------|--|
| <code>addis Rd, Rs1, constant</code> | $Rd := Rs1 + (constant \ll 16)$ <i>d</i> and <i>s1</i> are register numbers in the range 0..31. |

B.4 and, and.

The `and` instruction requires three register operands—a destination register and two source registers. This instruction computes the logical (bitwise) AND of the two source values and places the result in the destination register.

Table B-6: Gas Syntax for and

| Instruction | Description |
|--|--|
| and <i>Rd</i> , <i>Rs1</i> , <i>Rs2</i> | <i>Rd</i> := <i>Rs1</i> AND <i>Rs2</i> <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| and. <i>Rd</i> , <i>Rs1</i> , <i>Rs2</i> | <i>Rd</i> := <i>Rs1</i> AND <i>Rs2</i> CRO reflects the result of the operation. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |

Table B-7: CRO Settings for and.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | Unaffected |

B.5 andc, andc.

The andc instruction requires three register operands—a destination register and two source registers. This instruction computes the logical (bitwise) AND of the first source value with the inverted value of the second source operand and places the result in the destination register.

Table B-8: Gas Syntax for andc

| Instruction | Description |
|---|---|
| andc <i>Rd</i> , <i>Rs1</i> , <i>Rs2</i> | <i>Rd</i> := <i>Rs1</i> AND (NOT <i>Rs2</i>) <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| andc. <i>Rd</i> , <i>Rs1</i> , <i>Rs2</i> | <i>Rd</i> := <i>Rs1</i> AND (NOT <i>Rs2</i>) CRO reflects the result of the operation. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |

Table B-9: CRO Settings for andc.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | Unaffected |

B.6 andi

The `andi` (and immediate) instruction requires two register operands and a 16-bit constant. This instruction computes the logical (bitwise) AND of the value in the second (source) register and the constant value and places the result in the first (destination) register. Note that this instruction always clears the HO bits of the destination register.

Table B-10: Gas Syntax for `andi`

| Instruction | Description |
|------------------------------------|--|
| <code>andi Rd, Rs, constant</code> | $Rd := Rs \text{ AND } constant$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. |

B.7 andis

The `andis` (and immediate, shifted) instruction requires two register operands and a 16-bit constant. This instruction shifts the constant to the left 16 bits, logically ANDs this with the value held in the source register, and then places the result in the destination register. Note that this instruction always clears the LO bits of the destination register.

Table B-11: Gas Syntax for `andis`

| Instruction | Description |
|-------------------------------------|---|
| <code>andis Rd, Rs, constant</code> | $Rd := Rs \text{ AND } (constant \ll 16)$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. |

B.8 Branches

Standard PowerPC assembly language exposes the numeric encoding of the opcode in the standard branch mnemonics. If you're reading arbitrary PowerPC assembly code, you might have to memorize "magic numbers" that appear in the operand field of various branch instructions. Fortunately, IBM has defined a set of "mnemonic synonyms" that use English names for various numeric encodings. Compilers like GCC typically use the synonyms rather than the numeric forms. In this appendix, I'll discuss these "simplified branch mnemonics." If you encounter weird forms of the branch instructions, you may want to consult the PowerPC programmer's reference guide (i.e., PowerPC Microprocessor Family: The Programmer's Reference Guide) for their exact interpretation.

The PowerPC branch instructions provide four basic addressing modes: relative, absolute, indirect through LINK, and indirect through COUNT. GCC doesn't seem to use the absolute addressing mode (it's useful mainly in embedded systems where you have good control over the memory map), so I'll not consider that form here.

B.8.1 Unconditional Branch (b), Relative

The branch relative instruction encodes a 24-bit relative displacement field as part of the opcode. The CPU shifts this 24-bit value to the left two positions (producing a 26-bit value), sign extends the result to 32 bits, and then adds this displacement to the CPU's program counter register (CIA, or current instruction address, on the PowerPC).

Table B-12: Gas Syntax for b

| Instruction | Description |
|------------------|---|
| b target_address | $NIA := CIA + displacement$ NIA is the next instruction address. CIA is the current instruction address. <i>displacement</i> is the distance from the current instruction to the target_address. |

B.8.2 Unconditional Branch and Link (bl), Relative

The bl (branch and link) instruction operates almost identically to the unconditional branch instruction. The only difference is that in addition to transferring control, it also copies the address of the next instruction (after the branch) into the LINK register. Programs generally use the bl instruction to call local subroutines.

Table B-13: Gas Syntax for bl

| Instruction | Description |
|-------------------|--|
| bl target_address | $LINK := CIA + 4$ $NIA := CIA + displacement$ NIA is the next instruction address. CIA is the current instruction address. <i>displacement</i> is the distance from the current instruction to the target_address. |

B.8.3 Indirect Branch Instructions (blr and bctr)

The PowerPC provides two instructions that transfer control to an address held in either the LINK or COUNT register. The blr (branch to link register) instruction is typically used to return control from some subroutine. The bctr instruction is a general-purpose indirect branch that a compiler can use to implement control statements like C's switch statement.

Table B-14: Gas Syntax for blr and bctr

| Instruction | Description |
|-------------|--|
| blr | $NIA := LINK$ NIA is the next instruction address. |
| bctr | $NIA := COUNT$ NIA is the next instruction address. |

B.8.4 Conditional Branch Instructions

The PowerPC provides a wide range of conditional branch instructions that support the same addressing modes as the unconditional branches (relative, absolute, indirect through LINK, and indirect through COUNT). There are also forms that will save the address of the next instruction in the LINK register. The raw form of these conditional branch instructions allow you to test the condition bits found in any of the eight PowerPC condition-code registers (CR0..CR7). However, most assemblers (like Gas) provide “simplified mnemonics” that let you test a specific condition in CR0. As these are the branch instructions you’ll see used most often, we’ll discuss those forms here. For details on the other forms, see the PowerPC Microprocessor Family: The Programmer’s Reference Guide.

The conditional branches only support a 16-bit displacement (14 bits extended to 16 bits, actually). Therefore, the range of the conditional branches is substantially less than the unconditional branches (plus or minus 32,768 bytes). This generally isn’t much of a problem because conditional branches typically do not transfer control over great distances in typical programs.

Table B-15: Gas Syntax for Conditional Branches

| Instruction | Description |
|--------------------|---|
| blt target | Branch if less than. If the LT bit in CR0 is set, then add the 16-bit displacement to the current instruction address (CIA) to obtain the next instruction address (NIA). Otherwise, set the NIA to CIA+4. |
| ble target | Branch if less than or equal. If the LT or EQ bit in CR0 is set, then add the 16-bit displacement to the current instruction address (CIA) to obtain the next instruction address (NIA). Otherwise, set the NIA to CIA+4. |
| beq target | Branch if equal. If the EQ bit in CR0 is set, then add the 16-bit displacement to the current instruction address (CIA) to obtain the next instruction address (NIA). Otherwise, set the NIA to CIA+4. |
| bgt target | Branch if greater than. If the GT bit in CR0 is set, then add the 16-bit displacement to the current instruction address (CIA) to obtain the next instruction address (NIA). Otherwise, set the NIA to CIA+4. |
| bge target | Branch if greater than or equal. If the GT or EQ bit in CR0 is set, then add the 16-bit displacement to the current instruction address (CIA) to obtain the next instruction address (NIA). Otherwise, set the NIA to CIA+4. |
| bnl target | Branch if not less than. Synonym for bge. |
| bne target | Branch if not equal. If the EQ bit in CR0 is clear, then add the 16-bit displacement to the current instruction address (CIA) to obtain the next instruction address (NIA). Otherwise, set the NIA to CIA+4. |

Table B-15: Gas Syntax for Conditional Branches (continued)

| Instruction | Description |
|--------------------|--|
| bng target | Branch if not greater than. Synonym for ble. |
| bso target | Branch if summary overflow. If the SO bit in CRO is set, then add the 16-bit displacement to the current instruction address (CIA) to obtain the next instruction address (NIA). Otherwise, set the NIA to CIA+4. |
| bns target | Branch if not summary overflow. If the SO bit in CRO is clear, then add the 16-bit displacement to the current instruction address (CIA) to obtain the next instruction address (NIA). Otherwise, set the NIA to CIA+4. |

B.8.5 Indirect Conditional Branches

In addition to the relative conditional branches, the PowerPC also supports indirect versions that transfer control to the address held in the LINK or COUNT register. These instructions do not have any operands (as the LINK or COUNT register specifies the target address) and use the syntax shown here.

Table B-16: Indirect Conditional Branches

| Indirect Branch | | |
|------------------------|--------------|---|
| LINK | COUNT | Description |
| bltlr | bltctr | Branch if less than, indirect. |
| blelr | blectr | Branch if less than or equal, indirect. |
| beqlr | beqctr | Branch if equal, indirect. |
| bgtlr | bgtctr | Branch if greater than, indirect. |
| bgelr | bgectr | Branch if greater than or equal. |
| bnllr | bnlctr | Branch if not less than. Synonym for bge. |
| bnelr | bnectr | Branch if not equal. |
| bnglr | bngctr | Branch if not greater than. |
| bsolr | bsoctr | Branch if summary overflow. |
| bnslr | bnsctr | Branch if not summary overflow. |

B.8.6 Other Branch Forms

The PowerPC provides a bewildering array of options on the branch instructions. Not many of those other forms are used in this book, so there is no need to consider them here. Please consult the PowerPC Microprocessor Family: The Programmer's Reference Guide for more details on the available forms of the branch instructions.

B.9 cmp

The `cmp` instruction compares the *signed* values in two registers and updates the bits in one of the condition-code registers to reflect the comparison's results. By default, the `cmp` instruction assumes that you want to use CR0 to hold the result, though it is possible to specify a different condition-code register as the target for the comparison operation.

The `cmp` instruction sets the LT bit in the condition-code register if the first operand is less than the second operation (using a signed comparison). It sets the GT bit if the first operand is greater than the second. It sets the EQ bit if the two register operands hold the same value. This instruction also copies the summary overflow bit from the XER register into the SO bit of the condition-code register.

Table B-17: Gas Syntax for `cmp`

| Instruction | Description |
|---------------------------|--|
| <code>cmp Rs1, Rs2</code> | CR0 := Rs1 CMP Rs2 <i>s1</i> and <i>s2</i> are register numbers in the range 0..31. |

Table B-18: CR0 Settings for `cmp`

| Flag | Setting |
|------|--|
| LT | Set if the value in <i>Rs1</i> (signed) is less than <i>Rs2</i> . |
| GT | Set if the value in <i>Rs1</i> (signed) is greater than <i>Rs2</i> . |
| Zero | Set values in <i>Rs1</i> and <i>Rs2</i> are equal. |
| SO | Copied from the SO bit in the XER register. |

B.10 cmpi

The `cmpi` (compare immediate) instruction compares the *signed* value in a register against a constant and updates the bits in one of the condition-code registers. By default, the `cmpi` instruction assumes that you want to use CR0 to hold the result, though it is possible to specify a different condition-code register as the target for the comparison operation.

Table B-19: Gas Syntax for `cmpi`

| Instruction | Description |
|--------------------------------|--|
| <code>cmpi Rs, constant</code> | CR0 := Rs CMP <i>constant</i> <i>s</i> is a register number in the range 0..31. <i>constant</i> is a 16-bit signed constant. |

Table B-20: CRO Settings for `cmpi`

| Flag | Setting |
|------|--|
| LT | Set if the value in <i>Rs1</i> (signed) is less than <i>constant</i> . |
| GT | Set <i>Rs</i> 's value (signed) is greater than <i>constant</i> . |
| Zero | Set value in <i>Rs1</i> is equal to <i>constant</i> . |
| SO | Copied from the SO bit in the XER register. |

B.11 `cmpl`

The `cmpl` (compare logical) instruction is similar to `cmp` except that it does an unsigned comparison rather than a signed comparison. The syntax and usage is the same (except, of course, that you use the `cmpl` mnemonic). See `cmp` for more details.

B.12 `cmpli`

The `cmpli` (compare logical immediate) instruction is similar to `cmpi` except it does an unsigned comparison. The syntax and usage is similar to `cmpi` except that you use the `cmpli` mnemonic and the 16-bit constant must be an unsigned value in the range 0..65,535. See the description of the `cmpi` instruction for more details.

B.13 `divw`, `divw.`, `divwo`, `divwo.`

The `divw` (divide word, signed) instruction divides the value in one register by the value in a second register and stores the signed quotient into a third register. The version with the period suffix updates CRO after the division operation by comparing the quotient against zero. The version with the `o` suffix updates the overflow flag if the division operation is illegal (e.g., a division by zero).

Table B-21: Gas Syntax for `divw`

| Instruction | Description |
|---------------------------------|--|
| <code>divw Rd, Rs1, Rs2</code> | $Rd := Rs1 / Rs2$ <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| <code>divw. Rd, Rs1, Rs2</code> | $Rd := Rs1 / Rs2$ CRO reflects the result of the quotient. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |

Table B-21: Gas Syntax for `divw` (continued)

| Instruction | Description |
|----------------------------------|---|
| <code>divwo Rd, Rs1, Rs2</code> | $Rd := Rs1 / Rs2$ The overflow and summary overflow bits in XER are set if an error occurs. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| <code>divwo. Rd, Rs1, Rs2</code> | $Rd := Rs1 / Rs2$ CRO reflects the result of the quotient. The overflow and summary overflow bits in XER are set if an error occurs. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |

Table B-22: CRO Settings for `divw.` and `divwo.`

| Flag | Setting |
|------|--|
| LT | Set if the quotient (signed) is less than zero. |
| GT | Set if the quotient (signed) is greater than zero. |
| Zero | Set if the quotient is zero. |
| SO | The summary overflow bit from the XER is copied to this field after computing the sum. |

Table B-23: XER Settings for `divwo` and `divwo.`

| Flag | Setting |
|------|--|
| OV | Set if an error (division by zero or overflow) occurred during the execution of the instruction. |
| SO | Set if the SO bit was previously set, or if a division error occurred during the execution of the instruction. |
| CA | Unaffected. |

B.14 `divwu`, `divwu.`, `divwuo`, `divwuo.`

The `divwu` (divide word, unsigned) instruction divides the value in one register by the value in a second register and stores the unsigned quotient in a third register. The version with the period suffix updates CRO after the division operation by comparing the quotient against zero. The version with the `o` suffix updates the overflow flag if the division operation is illegal (e.g., a division by zero).

Table B-24: Gas Syntax for `divwu`

| Instruction | Description |
|----------------------------------|--|
| <code>divwu Rd, Rs1, Rs2</code> | $Rd := Rs1 / Rs2$ <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| <code>divwu. Rd, Rs1, Rs2</code> | $Rd := Rs1 / Rs2$ CRO reflects the result of the quotient. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |

Table B-24: Gas Syntax for `divwu` (continued)

| Instruction | Description |
|----------------------------------|--|
| <code>divwu Rd, Rs1, Rs2</code> | $Rd := Rs1 / Rs2$ The overflow and summary overflow bits in XER are set if an error occurs. d , $s1$, and $s2$ are register numbers in the range 0..31. |
| <code>divwu. Rd, Rs1, Rs2</code> | $Rd := Rs1 / Rs2$ CRO reflects the result of the quotient. The overflow and summary overflow bits in XER are set if an error occurs. d , $s1$, and $s2$ are register numbers in the range 0..31. |

Table B-25: CRO Settings for `divwu.` and `divwu.`

| Flag | Setting |
|------|--|
| LT | Set if the quotient is less than zero. |
| GT | Set if the quotient is greater than zero. |
| Zero | Set if the quotient is zero. |
| SO | The summary overflow bit from the XER is copied to this field after computing the sum. |

Table B-26: XER Settings for `divwu` and `divwu.`

| Flag | Setting |
|------|--|
| OV | Set if an error (division by zero or overflow) occurred during the execution of the instruction. |
| SO | Set if the SO bit was previously set, or if a division error occurred during the execution of the instruction. |
| CA | Unaffected. |

B.15 `equ, equ.`

The `equ` instruction requires three register operands—a destination register and two source registers. This instruction computes the logical XNOR of the two source values and places the result in the destination register. XNOR is also known as the “equals” function, hence the mnemonic. The `equ` instruction performs a bit-by-bit comparison of two 32-bit values. It stores a one in the corresponding destination bit position of the two source bit values are equal, it stores a zero in the destination bit position of the two source bits are not equal.

Table B-27: Gas Syntax for `equ`

| Instruction | Description |
|--------------------------------|--|
| <code>equ Rd, Rs1, Rs2</code> | $Rd := Rs1 == Rs2$ <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| <code>equ. Rd, Rs1, Rs2</code> | $Rd := Rs1 == Rs2$ CRO reflects the result of the operation. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |

Table B-28: CRO Settings for `equ`.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | Unaffected |

B.16 `extsb, extsb.`

The `extsb` instruction sign extends an 8-bit value to 32 bits. This instruction requires two register operands—a source register and a destination register. It extracts the byte from the LO 8 bits of the first register, sign extends the value to 32 bits, and then stores the result into the destination register.

Table B-29: Gas Syntax for `extsb`

| Instruction | Description |
|----------------------------|---|
| <code>extsb Rd, Rs</code> | $Rd := \text{signExtend}(Rs[0..7])$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. |
| <code>extsb. Rd, Rs</code> | $Rd := \text{signExtend}(Rs[0..7])$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. |

Table B-30: CRO Settings for `extsb`.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | Unaffected |

B.17 `extsh, extsh.`

The `extsh` instruction sign extends a 16-bit (halfword) value to 32 bits. This instruction requires two register operands—a source register and a destination register. It extracts the halfword from the LO 16 bits of the first register, sign extends the value to 32 bits, and then stores the result into the destination register.

Table B-31: Gas Syntax for extsh

| Instruction | Description |
|------------------------------|--|
| extsh <i>Rd</i> , <i>Rs</i> | $Rd := \text{signExtend}(Rs[0..15])$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. |
| extsh. <i>Rd</i> , <i>Rs</i> | $Rd := \text{signExtend}(Rs[0..15])$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. |

Table B-32: CRO Settings for extsh.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | Unaffected |

B.18 la

The `la` (load address) instruction is a synonym for the `addi` instruction. This instruction computes the effective address of a register plus displacement addressing mode and places the address in a destination register.

Table B-33: Gas Syntax for la

| Instruction | Description |
|--|---|
| la <i>Rd</i> , <i>disp</i> (<i>Rs</i>) | $Rd := \text{constant} + Rs$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. This instruction is equivalent to: <code>addi <i>Rd</i>, <i>Rs</i>, <i>constant</i></code> |

B.19 lbz, lbzu, lbzux, lbzx

The `lbz` (load byte and zero) instruction fetches a byte from memory at an address specified by the register plus displacement addressing mode. The `lbz` instruction zero extends this 8-bit value to 32 bits and stores the result in the destination register.

The `lbzu` (load byte and zero, with update) works in a similar manner except that it also updates the base address register with the effective address of the byte in memory.

The `lbzx` (load byte and zero, indexed) also zero extends an 8-bit value in memory to 32 bits and loads this result into a destination register. This form of the instruction, however, uses both a base and index register (with no displacement).

The `lbzux` (load byte and zero, indexed, with update) is just like `lbzx` except it also updates the base register with the effective address after moving the byte into the destination register.

Table B-34: Gas Syntax for `lbz`

| Instruction | Description |
|----------------------------------|--|
| <code>lbz Rd, disp(Rs)</code> | $Rd := \text{zeroExtend}(\text{mem8}[\text{disp} + Rs])$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. <i>disp</i> is a 16-bit signed constant. $\text{mem8}[--]$ is the byte at the memory address specified by <i>disp</i> + <i>Rs</i> . If <i>Rs</i> is RO, then this instruction substitutes the value zero for RO. |
| <code>lbzu Rd, disp(Rs)</code> | $Rd := \text{zeroExtend}(\text{mem8}[\text{disp} + Rs])$ $Rs := \text{disp} + Rs$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. <i>disp</i> is a 16-bit signed constant. If <i>Rs</i> is RO, or <i>Rs</i> and <i>Rd</i> are the same, this is an invalid instruction. |
| <code>lbzx Rd, Rs, Rx</code> | $Rd := \text{zeroExtend}(\text{mem8}[Rs + Rx])$ <i>d</i> , <i>s</i> , and <i>x</i> are register numbers in the range 0..31. If <i>Rs</i> is RO, then this instruction uses zero as the value for <i>Rs</i> . |
| <code>lbzux Rd, Rs, Rx</code> | $Rd := \text{zeroExtend}(\text{mem8}[Rs + Rx])$ $Rs := Rs + Rx$ <i>d</i> , <i>s</i> , and <i>x</i> are register numbers in the range 0..31. If <i>Rs</i> is RO, or <i>Rs</i> and <i>Rd</i> are the same, this is an invalid instruction. |

B.20 `lha`, `lhau`, `lhax`, `lhaux`

The `lha` (load halfword, algebraic) instruction fetches a 16-bit word from memory at an address specified by the register plus displacement addressing mode. The `lha` instruction sign extends this 16-bit value to 32 bits and stores the result in the destination register.

The `lhau` (load halfword, algebraic, with update) works in a similar manner except that it also updates the base register with the effective address of the halfword in memory.

The `lhax` (load halfword, algebraic, indexed) also sign extends a 16-bit value in memory to 32 bits and loads this result into a destination register. This form of the instruction, however, uses both a base and index register (with no displacement).

The `lhaux` (load halfword, algebraic, indexed, with update) is just like `lhax` except it also updates the base register with the effective address after moving the halfword into the destination register.

Table B-35: Gas Syntax for lha

| Instruction | Description |
|--|--|
| lha <i>Rd</i> , <i>disp</i> (<i>Rs</i>) | $Rd := \text{signExtend}(\text{mem16}[\text{disp} + Rs])$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. <i>disp</i> is a 16-bit signed constant. <i>mem16</i> [--] is the 16-bit halfword at the memory address specified by <i>disp</i> + <i>Rs</i> . If <i>Rs</i> is R0, then this instruction substitutes the value zero for R0. |
| lhau <i>Rd</i> , <i>disp</i> (<i>Rs</i>) | $Rd := \text{signExtend}(\text{mem16}[\text{disp} + Rs])$ $Rs := \text{disp} + Rs$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. <i>disp</i> is a 16-bit signed constant. If <i>Rs</i> is R0, or <i>Rs</i> and <i>Rd</i> are the same, this is an invalid instruction. |
| lhax <i>Rd</i> , <i>Rs</i> , <i>Rx</i> | $Rd := \text{signExtend}(\text{mem16}[Rs + Rx])$ <i>d</i> , <i>s</i> , and <i>x</i> are register numbers in the range 0..31. If <i>Rs</i> is R0, then this instruction uses zero as the value for <i>Rs</i> . |
| lhaux <i>Rd</i> , <i>Rs</i> , <i>Rx</i> | $Rd := \text{signExtend}(\text{mem16}[Rs + Rx])$ $Rs := Rs + Rx$ <i>d</i> , <i>s</i> , and <i>x</i> are register numbers in the range 0..31. If <i>Rs</i> is R0, or <i>Rs</i> and <i>Rd</i> are the same, this is an invalid instruction. |

B.21 lhz, lhzu, lhzx, lhzux

The lhz (load halfword and zero) instruction fetches a 16-bit word from memory at an address specified by the register plus displacement addressing mode. The lhz instruction zero extends this 16-bit value to 32 bits and stores the result in the destination register.

The lhzu (load halfword and zero, with update) works in a similar manner except that it also updates the base register with the effective address of the halfword in memory.

The lhzx (load halfword and zero, indexed) also zero extends a 16-bit value in memory to 32 bits and loads this result into a destination register. This form of the instruction, however, uses both a base and index register (with no displacement).

The lhzux (load halfword and zero, indexed, with update) is just like lhzx except it also updates the base register with the effective address after moving the halfword into the destination register.

Table B-36: Gas Syntax for lhz

| Instruction | Description |
|--|--|
| lhz <i>Rd</i> , <i>disp</i> (<i>Rs</i>) | $Rd := \text{zeroExtend}(\text{mem16}[\text{disp} + Rs])$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. <i>disp</i> is a 16-bit signed constant. mem16[--] is the 16-bit halfword at the memory address specified by <i>disp</i> + <i>Rs</i> . If <i>Rs</i> is R0, then this instruction substitutes the value zero for R0. |
| lhzu <i>Rd</i> , <i>disp</i> (<i>Rs</i>) | $Rd := \text{zeroExtend}(\text{mem16}[\text{disp} + Rs])$ $Rs := \text{disp} + Rs$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. <i>disp</i> is a 16-bit signed constant. If <i>Rs</i> is R0, or <i>Rs</i> and <i>Rd</i> are the same, this is an invalid instruction. |
| lhx <i>Rd</i> , <i>Rs</i> , <i>Rx</i> | $Rd := \text{zeroExtend}(\text{mem16}[Rs + Rx])$ <i>d</i> , <i>s</i> , and <i>x</i> are register numbers in the range 0..31. If <i>Rs</i> is R0, then this instruction uses zero as the value for <i>Rs</i> . |
| lhzux <i>Rd</i> , <i>Rs</i> , <i>Rx</i> | $Rd := \text{zeroExtend}(\text{mem16}[Rs + Rx])$ $Rs := Rs + Rx$ <i>d</i> , <i>s</i> , and <i>x</i> are register numbers in the range 0..31. If <i>Rs</i> is R0, or <i>Rs</i> and <i>Rd</i> are the same, this is an invalid instruction. |

B.22 li

The *li* (load immediate) instruction is a synonym for the *addi* instruction with R0 specified as the source register. This instruction loads a sign-extended 16-bit value into the specified destination register.

Table B-37: Gas Syntax for li

| Instruction | Description |
|--------------------------------|---|
| li <i>Rd</i> , <i>constant</i> | $Rd := \text{constant}$ <i>d</i> is a register number in the range 0..31. This instruction is equivalent to: <i>addi Rd</i> , 0, <i>constant</i> |

B.23 lis

The *lis* instruction (load immediate, shifted) shifts a 16-bit constant to the left 16 bits and then stores the value into a destination register. This instruction does not affect any flags or the overflow bit.

Table B-38: Gas Syntax for lis

| Instruction | Description |
|---------------------------------|---|
| lis <i>Rd</i> , <i>constant</i> | $Rd := (\text{constant} \ll 16)$ <i>d</i> is a register number in the range 0..31. This instruction is a synonym for: <i>addis Rd</i> , 0, <i>constant</i> |

B.24 lmw

The `lmw` (load multiple word) loads a group of registers from a contiguous block of memory. This instruction has two operands: a starting destination register and a register plus displacement effective memory address. This instruction loads all the registers from the destination register through R31 starting at the specified memory location. This instruction is quite useful for saving a batch of scratch-pad registers or for quickly moving blocks of memory around. Note that the base register used in the memory addressing mode must not be present in the range of registers loaded by this instruction.

Table B-39: Gas Syntax for `lmw`

| Instruction | Description |
|---------------------------------|--|
| <code>lmw Rd, disp(Rs)</code> | $Rd..R31 := \text{mem32}[\text{disp} + Rs]..$ d and s are register numbers in the range 0..31 and s must be less than d . disp is a 16-bit signed constant. $\text{mem32}[--]..$ represents n consecutive 32-bit words in memory, where $n = 31 - d + 1$ |

B.25 lwz, lwzu, lwzx, lwzux

The `lwz` (load word and zero) instruction fetches a 32-bit word from memory at an address specified by the register plus displacement addressing mode. (The `z` suffix exists for 64-bit members of the PowerPC family, in which case this instruction zero extends the memory value to 64 bits).

The `lwzu` (load word and zero, with update) works in a similar manner except that it also updates the source register with the effective address of the word in memory.

The `lwzx` (load word and zero, indexed) also loads a 32-bit value from memory into a destination register. This form of the instruction, however, uses both a base and index register (with no displacement).

The `lwzux` (load word and zero, indexed, with update) is just like `lwzx` except it also updates the base register with the effective address after moving the 32-bit word into the destination register.

Table B-40: Gas Syntax for `lww`

| Instruction | Description |
|----------------------------------|---|
| <code>lww Rd, disp(Rs)</code> | $Rd := \text{mem32}[disp + Rs]$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. <i>disp</i> is a 16-bit signed constant. $\text{mem32}[--]$ is the 32-bit word at the memory address specified by <i>disp</i> + <i>Rs</i> . If <i>Rs</i> is RO, then this instruction substitutes the value zero for RO. |
| <code>lwwz Rd, disp(Rs)</code> | $Rd := \text{mem32}[disp + Rs]$ $Rs := disp + Rs$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. <i>disp</i> is a 16-bit signed constant. If <i>Rs</i> is RO, or <i>Rs</i> and <i>Rd</i> are the same, this is an invalid instruction. |
| <code>lwsx Rd, Rs, Rx</code> | $Rd := \text{mem32}[Rs + Rx]$ <i>d</i> , <i>s</i> , and <i>x</i> are register numbers in the range 0..31. If <i>Rs</i> is RO, then this instruction uses zero as the value for <i>Rs</i> . |
| <code>lwwux Rd, Rs, Rx</code> | $Rd := \text{mem32}[Rs + Rx]$ $Rs := Rs + Rx$ <i>d</i> , <i>s</i> , and <i>x</i> are register numbers in the range 0..31. If <i>Rs</i> is RO, or <i>Rs</i> and <i>Rd</i> are the same, this is an invalid instruction. |

B.26 `mcrf`

The `mcrf` (move condition register field) instruction moves the data from one condition-code register field to another condition-code register field.

Table B-41: Gas Syntax for `mcrf`

| Instruction | Description |
|----------------------------|--|
| <code>mcrf CRd, CRs</code> | $CRd := CRs$ <i>d</i> and <i>s</i> are condition-code register numbers in the range 0..7. |

B.27 `mcrxr`

The `mcrxr` (move condition register field from XER) instruction copies bits 0..3 of the XER register (the SO, OV, and CA flags, along with a zero bit) into the specified condition-code register. This instruction also clears bits 0..3 of the XER register.

Table B-42: Gas Syntax for `mcrxr`

| Instruction | Description |
|------------------------|--|
| <code>mcrxr CRd</code> | $CRd := \text{XER}[0..3]$ <i>d</i> is a condition-code register number in the range 0..7. |

Table B-43: CRd Settings for `mcrxr`

| Flag | Setting |
|------|-------------------|
| LT | SO field from XER |
| GT | OV field from XER |
| Zero | CA field from XER |
| SO | 0 |

Table B-44: XER Settings for `mcrxr`

| Flag | Setting |
|------|---------|
| SO | 0 |
| OV | 0 |
| CA | 0 |

B.28 `mfcrr`

The `mfcrr` (move from condition register) instruction copies the entire 32-bit condition-code register into a general-purpose register.

Table B-45: Gas Syntax for `mfcrr`

| Instruction | Description |
|-----------------------|---|
| <code>mfcrr Rd</code> | $Rd := CR[0..7]$ <i>d</i> is a general-purpose register number in the range 0..31. |

B.29 `mfcrr`

The `mfcrr` (move from COUNT register) instruction copies the contents of the COUNT register into a general-purpose register.

Table B-46: Gas Syntax for `mfcrr`

| Instruction | Description |
|-----------------------|--|
| <code>mfcrr Rd</code> | $Rd := COUNT$ <i>d</i> is a general-purpose register number in the range 0..31. |

B.30 `mflr`

The `mflr` (move from LINK register) instruction copies the contents of the LINK register into a general-purpose register.

Table B-47: Gas Syntax for `mflr`

| Instruction | Description |
|----------------------|---|
| <code>mflr Rd</code> | $Rd := LINK$ <i>d</i> is a general-purpose register number in the range 0..31. |

B.31 `mr`

The `mr` instruction (move register) requires two register operands—a destination register and a source register. This instruction copies the value held in the source register to the destination register. Note that this is a special form of the `or` instruction that supplies the source register as both operands for the `or` instruction. See the `or` instruction for more details.

Table B-48: Gas Syntax for `mr`

| Instruction | Description |
|------------------------|--|
| <code>mr Rd, Rs</code> | $Rd := Rs$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. |

B.32 `mtcrf`

The `mtcrf` (move to condition register fields) instruction copies zero or more blocks of 4 bits into one of the condition-code fields in the condition-code register. This instruction has two operands: an 8-bit bitmap that specifies which condition-code fields to update and a general-purpose 32-bit register. For each set bit in the bitmap, this instruction copies the corresponding 4 bits in the general-purpose register to the corresponding positions in the condition-code register. If a bit in the bitmap contains zero, then the corresponding bits in the condition-code field are unaffected by this instruction.

Table B-49: Gas Syntax for `mtcrf`

| Instruction | Description |
|-------------------------------|--|
| <code>mtcrf bitmap, Rd</code> | $CRn := Rd[n*4..n*4+3]$, but only if $bitmap[n] == 1$ <i>d</i> is a general-purpose register number in the range 0..31. <i>bitmap</i> is an 8-bit constant. |

B.33 `mtctr`

The `mtctr` (move to COUNT) instruction copies the value from a general-purpose integer register to the COUNT register.

Table B-50: Gas Syntax for `mtctr`

| Instruction | Description |
|-----------------------|---|
| <code>mtctr Rd</code> | <code>COUNT := Rd</code> <i>d</i> is a general-purpose register number in the range 0..31. |

B.34 `mtlr`

The `mtlr` (move to LINK) instruction copies the value from a general-purpose integer register to the LINK register.

Table B-51: Gas Syntax for `mtlr`

| Instruction | Description |
|----------------------|--|
| <code>mtlr Rd</code> | <code>LINK := Rd</code> <i>d</i> is a general-purpose register number in the range 0..31. |

B.35 `mtxer`

The `mtxer` (move to XER) instruction copies the value from a general-purpose integer register to the XER register.

Table B-52: Gas Syntax for `mtxer`

| Instruction | Description |
|-----------------------|---|
| <code>mtxer Rd</code> | <code>XER := Rd</code> <i>d</i> is a general-purpose register number in the range 0..31. |

B.36 `mulhw`, `mulhw.`

The `mulhw` (multiply high word) instruction produces the HO 32 bits of a 32×32 multiply of two registers. It stores the HO 32 bits of the product in a third register. This instruction performs a signed integer multiplication.

Table B-53: Gas Syntax for `mulhw`

| Instruction | Description |
|----------------------------------|---|
| <code>mulhw Rd, Rs1, Rs2</code> | <code>Rd := H032(Rs1 × Rs2)</code> (signed) <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| <code>mulhw. Rd, Rs1, Rs2</code> | <code>Rd := H032(Rs1 × Rs2)</code> (signed) <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. This form updates CRO (see Table B-54). |

Table B-54: CRO Settings for mulhw.

| Flag | Setting |
|------|--|
| LT | Set if the signed result is less than zero. |
| GT | Set if the signed result is greater than zero. |
| Zero | Set if the result is equal to zero. |
| SO | Copied from the SO bit in the XER register. |

B.37 mulhwu, mulhwu.

The mulhwu (multiply high word, unsigned) instruction produces the HO 32 bits of an unsigned 32×32 multiply of two registers. It stores the HO 32 bits of the product in a third register.

Table B-55: Gas Syntax for mulhwu

| Instruction | Description |
|---|---|
| mulhwu <i>Rd</i> , <i>Rs1</i> , <i>Rs2</i> | $Rd := H032(Rs1 \times Rs2)$ (unsigned) <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| mulhwu. <i>Rd</i> , <i>Rs1</i> , <i>Rs2</i> | $Rd := H032(Rs1 \times Rs2)$ (unsigned) <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. This form updates CRO (see Table B-56). |

Table B-56: CRO Settings for mulhwu.

| Flag | Setting |
|------|--|
| LT | Set if the signed result is less than zero. |
| GT | Set if the signed result is greater than zero. |
| Zero | Set if the result is equal to zero. |
| SO | Copied from the SO bit in the XER register. |

B.38 mulli

The mulli (multiply low word, immediate) instruction produces the LO 32 bits of a 32×32 multiply of two registers. It stores the LO 32 bits of the product in a third register. Note that this instruction is suitable for both signed and unsigned operands as the LO 32 bits of the product is the same for both operand types.

Table B-57: Gas Syntax for mulli

| Instruction | Description |
|---|--|
| mulli <i>Rd</i> , <i>Rs</i> , <i>constant</i> | $Rd := Rs \times constant$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. <i>constant</i> is a 16-bit signed integer, which this instruction sign extends to 32 bits before the multiplication occurs. |

B.39 `mullw`, `mullw.`, `mullwo`, `mullwo.`

The `mullw` (multiply low word) instruction produces the LO 32 bits of a 32×32 multiplication of two registers. It stores the LO 32 bits of the product in a third register. The LO 32 bits of a 32×32 multiplication is the same for both signed and unsigned multiplications, so you'd use this instruction to compute the result for either type of data.

Table B-58: Gas Syntax for `mullw`

| Instruction | Description |
|-----------------------------------|--|
| <code>mullw Rd, Rs1, Rs2</code> | $Rd := Rs1 \times Rs2$ (LO 32 bits) <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| <code>mullwo Rd, Rs1, Rs2</code> | $Rd := Rs1 \times Rs2$ (LO 32 bits) <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. This form updates XER. |
| <code>mullw. Rd, Rs1, Rs2</code> | $Rd := Rs1 \times Rs2$ (LO 32 bits) <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. This form updates CRO. |
| <code>mullwo. Rd, Rs1, Rs2</code> | $Rd := Rs1 \times Rs2$ (LO 32 bits) <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. This form updates XER and CRO. |

Table B-59: CRO Settings for `mullw.`, `mullwo.`

| Flag | Setting |
|------|--|
| LT | Set if the signed result is less than zero. |
| GT | Set if the signed result is greater than zero. |
| Zero | Set if the result is equal to zero. |
| SO | Copied from the SO bit in the XER register. |

Table B-60: XER Settings for `mullwo`, `mullwo.`

| Flag | Setting |
|------|---|
| SO | Set if SO was previously set, or the signed result does not fit into 32 bits. |
| OV | Set if the signed result does not fit into 32 bits. |
| CA | Unaffected. |

B.40 `nand`, `nand`.

The `nand` instruction requires three register operands—a destination register and two source registers. This instruction computes the logical (bitwise) NAND (NOT AND) of the two source values and places the result in the destination register.

Table B-61: Gas Syntax for nand

| Instruction | Description |
|---|---|
| nand <i>Rd</i> , <i>Rs1</i> , <i>Rs2</i> | <i>Rd</i> := <i>Rs1</i> NAND <i>Rs2</i> <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| nand. <i>Rd</i> , <i>Rs1</i> , <i>Rs2</i> | <i>Rd</i> := <i>Rs1</i> NAND <i>Rs2</i> CRO reflects the result of the operation. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |

Table B-62: CRO Settings for nand.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | Unaffected. |

B.41 neg, neg., nego, nego.

The neg instruction requires two register operands—a destination register and a source register. This instruction computes the two's complement of the value in the source register (that is, it negates the value) and places the result into the destination register.

Table B-63: Gas Syntax for neg

| Instruction | Description |
|-----------------------------|---|
| neg <i>Rd</i> , <i>Rs</i> | <i>Rd</i> := - <i>Rs</i> <i>d</i> and <i>s</i> are register numbers in the range 0..31. |
| neg. <i>Rd</i> , <i>Rs</i> | <i>Rd</i> := - <i>Rs</i> CRO reflects the result of the negation. <i>d</i> and <i>s</i> are register numbers in the range 0..31. |
| nego <i>Rd</i> , <i>Rs</i> | <i>Rd</i> := - <i>Rs</i> The overflow and summary overflow bits in XER are set if a signed overflow occurs (this occurs if you attempt to negate the most negative value). <i>d</i> and <i>s</i> are register numbers in the range 0..31. |
| nego. <i>Rd</i> , <i>Rs</i> | <i>Rd</i> := <i>Rs</i> CRO reflects the result of the sum. The overflow and summary overflow bits in XER are set if a signed overflow occurs. <i>d</i> and <i>s</i> are register numbers in the range 0..31. |

Table B-64: CRO Settings for neg. and nego.

| Flag | Setting |
|------|---|
| LT | Set if the result is less than zero. |
| GT | Set if the result is greater than zero. |

Table B-64: CRO Settings for neg. and nego. (continued)

| Flag | Setting |
|------|--|
| Zero | Set if the result is zero. |
| SO | The summary overflow bit from the XER is copied to this field after computing the sum. |

Table B-65: XER Settings for nego and nego.

| Flag | Setting |
|------|---|
| OV | Set if a signed overflow occurred during the execution of the instruction. This occurs if you attempt to negate the most negative value in the two's complement system (\$8000_0000 for 32-bit values). |
| SO | Set if the SO bit was previously set, or if a signed overflow occurred during the execution of the instruction. |
| CA | Unaffected. |

B.42 nor, nor.

The `nor` instruction requires three register operands—a destination register and two source registers. This instruction computes the logical (bitwise) NOR (NOT OR) of the two source values and places the result in the destination register. If both source operands are the same register, this instruction computes the logical NOT operation of that register.

Table B-66: Gas Syntax for `nor`

| Instruction | Description |
|--------------------------------|--|
| <code>nor Rd, Rs1, Rs2</code> | $Rd := Rs1 \text{ NOR } Rs2$ <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| <code>nor. Rd, Rs1, Rs2</code> | $Rd := Rs1 \text{ NOR } Rs2$ CRO reflects the result of the operation. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |

Table B-67: CRO Settings for `nor`.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | Unaffected. |

B.43 or, or.

The `or` instruction requires three register operands—a destination register and two source registers. This instruction computes the logical (bitwise) OR of the two source values and places the result in the destination register. If both source operands are the same register, this instruction is a synonym for the `mr` (move register) instruction (see `mr` for more details).

Table B-68: Gas Syntax for `or`

| Instruction | Description |
|-------------------------------|--|
| <code>or Rd, Rs1, Rs2</code> | $Rd := Rs1 \text{ OR } Rs2$ $d, s1,$ and $s2$ are register numbers in the range 0..31. |
| <code>or. Rd, Rs1, Rs2</code> | $Rd := Rs1 \text{ OR } Rs2$ CRO reflects the result of the operation. $d, s1,$ and $s2$ are register numbers in the range 0..31. |

Table B-69: CRO Settings for `or`.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | Unaffected. |

B.44 orc, orc.

The `orc` instruction requires three register operands—a destination register and two source registers. This instruction computes the logical (bitwise) OR of the first source value with the inverted value of the second source operand and places the result in the destination register.

Table B-70: Gas Syntax for `orc`

| Instruction | Description |
|--------------------------------|--|
| <code>orc Rd, Rs1, Rs2</code> | $Rd := Rs1 \text{ OR } (\text{NOT } Rs2)$ $d, s1,$ and $s2$ are register numbers in the range 0..31. |
| <code>orc. Rd, Rs1, Rs2</code> | $Rd := Rs1 \text{ OR } (\text{NOT } Rs2)$ CRO reflects the result of the operation. $d, s1,$ and $s2$ are register numbers in the range 0..31. |

Table B-71: CRO Settings for `orc`.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |

Table B-71: CRO Settings for `orc`. (continued)

| Flag | Setting |
|------|-------------------------|
| Zero | Set if the sum is zero. |
| SO | Unaffected |

B.45 `ori`

The `ori` (or immediate) instruction requires two register operands and a 16-bit constant. This instruction logically ORs the constant with the value held in the source register, and then places the result in the destination register.

Table B-72: Gas Syntax for `ori`

| Instruction | Description |
|-----------------------------------|---|
| <code>ori Rd, Rs, constant</code> | $Rd := Rs \text{ OR } constant$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. |

B.46 `oris`

The `oris` (or immediate, shifted) instruction requires two register operands and a 16-bit constant. This instruction shifts the constant to the left 16 bits, logically ORs this with the value held in the source register, and then places the result in the destination register.

Table B-73: Gas Syntax for `oris`

| Instruction | Description |
|------------------------------------|--|
| <code>oris Rd, Rs, constant</code> | $Rd := Rs \text{ OR } (constant \ll 16)$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. |

B.47 `rlwimi`, `rlwimi`.

The `rlwimi` instruction (rotate left word immediate, then mask insert) requires five operands—a destination register, a source register, and three immediate operands. This instruction rotates the source operand to the left by the number of bits specified by its first immediate operand (the third operand), and then extracts bits *mb..me* (the second and third immediate operands) from this result and inserts those bits into the destination register (without affecting the bits outside the range *mb..me* in the destination register).

Table B-74: Gas Syntax for `rlwimi`

| Instruction | Description |
|------------------------------------|--|
| <code>rlwimi Rd,Rs,n,mb,me</code> | $Rd := (Rd \text{ AND } \text{mask0}(mb..me)) \text{ OR } ((Rd \text{ ROL } n) \text{ AND } \text{mask1}(mb..me))$ n is a constant specifying the number of bits to rotate in the source register. mb and me specify the beginning and ending bit positions for the mask. $\text{mask0}(a..b)$ is a set of zero bits in positions $a..b$ and ones everywhere else. $\text{mask1}(a..b)$ is a set of one bits in positions $a..b$ and zeros everywhere else. d and s are register numbers in the range 0..31. |
| <code>rlwimi. Rd,Rs,n,mb,me</code> | $Rd := (Rd \text{ AND } \text{mask0}(mb..me)) \text{ OR } ((Rd \text{ ROL } n) \text{ AND } \text{mask1}(mb..me))$ CRO reflects the result of the operation. n is a constant specifying the number of bits to rotate in the source register. mb and me specify the beginning and ending bit positions for the mask. $\text{mask0}(a..b)$ is a set of zero bits in positions $a..b$ and ones everywhere else. $\text{mask1}(a..b)$ is a set of one bits in positions $a..b$ and zeros everywhere else. d and s are register numbers in the range 0..31. |

Table B-75: CRO Settings for `rlwimi`.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | Unaffected |

B.48 `rlwinm`, `rlwinm`.

The `rlwinm` instruction (rotate left word immediate, then AND with mask) requires five operands—a destination register, a source register, and three immediate operands. This instruction rotates the source operand to the left by the number of bits specified by its first immediate operand (the third operand), and then extracts bits $mb..me$ (the second and third immediate operands) from this result stores the result into the destination register (with zeros in bit positions outside the mask range).

Table B-76: Gas Syntax for `rlwinm`

| Instruction | Description |
|------------------------------------|---|
| <code>rlwinm Rd,Rs,n,mb,me</code> | $Rd := (Rd \text{ ROL } n) \text{ AND mask}(mb..me)$ <i>n</i> is a constant specifying the number of bits to rotate in the source register. <i>mb</i> and <i>me</i> specify the beginning and ending bit positions for the mask. <code>mask(a..b)</code> is a set of one bits in positions <i>a..b</i> and zeros everywhere else. <i>d</i> and <i>s</i> are register numbers in the range 0..31. |
| <code>rlwinm. Rd,Rs,n,mb,me</code> | $Rd := (Rd \text{ ROL } n) \text{ AND mask}(mb..me)$ CRO reflects the result of the operation. <i>n</i> is a constant specifying the number of bits to rotate in the source register. <i>mb</i> and <i>me</i> specify the beginning and ending bit positions for the mask. <code>mask(a..b)</code> is a set of one bits in positions <i>a..b</i> and zeros everywhere else. <i>d</i> and <i>s</i> are register numbers in the range 0..31. |

Table B-77: CRO Settings for `rlwinm`.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | Unaffected |

B.49 `rlwnm`, `rlwnm`.

The `rlwnm` instruction (rotate left word then AND with mask) requires five operands—a destination register, a source register, a register holding a count value, and two immediate operands. This instruction rotates the source operand to the left by the number of bits specified by count register operand (the third operand), and then extracts bits *mb..me* (the second and third immediate operands) from this result stores the result into the destination register (with zeros in bit positions outside the mask range).

Table B-78: Gas Syntax for `r1lwm`

| Instruction | Description |
|------------------------------------|---|
| <code>r1lwm Rd,Rs,Rc,mb,me</code> | $Rd := (Rd \text{ ROL } Rc) \text{ AND mask}(mb..me)$ mb and me specify the beginning and ending bit positions for the mask. $\text{mask}(a..b)$ is a set of one bits in positions $a..b$ and zeros everywhere else. d , s , and c are register numbers in the range 0..31. |
| <code>r1lwm. Rd,Rs,Rc,mb,me</code> | $Rd := (Rd \text{ ROL } Rc) \text{ AND mask}(mb..me)$ CRO reflects the result of the operation. mb and me specify the beginning and ending bit positions for the mask. $\text{mask}(a..b)$ is a set of one bits in positions $a..b$ and zeros everywhere else. d and s are register numbers in the range 0..31. |

Table B-79: CRO Settings for `r1lwm`.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | Unaffected. |

B.50 `slw`, `slw.`

The `slw` instruction (shift left word) requires three register operands—a destination register, a source register, a register holding a count value. This instruction shifts the value of the source operand to the left by the number of bits specified by the count register operand and stores the result into the destination register. This is an unsigned, or logical, shift left operation. Zeros are shifted into unoccupied LO bit positions. Bits shifted out of the HO bit are lost.

Table B-80: Gas Syntax for `slw`

| Instruction | Description |
|----------------------------|---|
| <code>slw Rd,Rs,Rc</code> | $Rd := (Rs \text{ SHL } Rc)$ d , s , and c are register numbers in the range 0..31. |
| <code>slw. Rd,Rs,Rc</code> | $Rd := (Rs \text{ SHL } Rc)$ CRO reflects the result of the operation. d , s , and c are register numbers in the range 0..31. |

Table B-81: CRO Settings for `slw`.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |

Table B-81: CRO Settings for `s1w`. (continued)

| Flag | Setting |
|------|-------------------------|
| Zero | Set if the sum is zero. |
| SO | Unaffected. |

B.51 `sraw`, `sraw`.

The `sraw` instruction (shift right, arithmetic, word) requires three register operands—a destination register, a source register, a register holding a count value. This instruction shifts the value of the source operand to the right by the number of bits specified by the count register operand and stores the result into the destination register. This instruction replicates the HO (sign) bit into the HO bit position after the shift. Bits shifted out of the LO bit position are lost.

Table B-82: Gas Syntax for `sraw`

| Instruction | Description |
|-----------------------------|--|
| <code>sraw Rd,Rs,Rc</code> | $Rd := (Rs \text{ SHR } Rc)$ (signed) d , s , and c are register numbers in the range 0..31. |
| <code>sraw. Rd,Rs,Rc</code> | $Rd := (Rs \text{ SHR } Rc)$ (signed) CRO reflects the result of the operation. d , s , and c are register numbers in the range 0..31. |

Table B-83: CRO Settings for `sraw`.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | Unaffected |

B.52 `srawi`, `srawi`.

The `srawi` instruction (shift right arithmetic word, immediate) requires two register operands (destination and source) and an immediate *count* value. This instruction shifts the value of the source operand to the right *count* bits and stores the result into the destination register. This instruction replicates the HO (sign) bit into the HO bit position after the shift. Bits shifted out of the LO bit position are lost.

Table B-84: Gas Syntax for `srawi`

| Instruction | Description |
|------------------------------------|---|
| <code>srawi Rd,Rs,constant</code> | $Rd := (Rs \text{ SHR } constant)$ (signed) <i>constant</i> is the number of bits to shift, in the range 0..31. <i>d</i> and <i>s</i> are register numbers in the range 0..31. |
| <code>srawi. Rd,Rs,constant</code> | $Rd := (Rs \text{ SHR } constant)$ (signed) CRO reflects the result of the operation. <i>constant</i> is the number of bits to shift, in the range 0..31. <i>d</i> and <i>s</i> are register numbers in the range 0..31. |

Table B-85: CRO Settings for `srawi`.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | Unaffected. |

B.53 `srw, srw.`

The `srw` instruction (shift right word) requires three register operands—a destination register, a source register, a register holding a count value. This instruction shifts the value of the source operand to the right by the number of bits specified by the count register and stores the result into the destination register. This is an unsigned, or logical, shift right operation. It shifts zeros into the unoccupied HO bit positions. Bits shifted out of the LO bit position are lost.

Table B-86: Gas Syntax for `srw`

| Instruction | Description |
|----------------------------|--|
| <code>srw Rd,Rs,Rc</code> | $Rd := (Rs \text{ SHL } Rc)$ <i>d</i> , <i>s</i> , and <i>c</i> are register numbers in the range 0..31. |
| <code>srw. Rd,Rs,Rc</code> | $Rd := (Rs \text{ SHL } Rc)$ CRO reflects the result of the operation. <i>d</i> , <i>s</i> , and <i>c</i> are register numbers in the range 0..31. |

Table B-87: CRO Settings for `srw`.

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | Unaffected. |

B.54 stb, stbu, stbux, stbx

The `stb` (store byte) instruction stores the LO byte of a register into memory at an address specified by the register plus displacement addressing mode.

The `stbu` (store byte with update) works in a similar manner except that it also updates the base register with the effective address of the byte in memory.

The `stbx` (store byte, indexed) stores the byte held in the LO byte of a source register into the memory location specified by the register plus register indexed addressing mode.

The `stbux` (store byte indexed, with update) is just like `stbx` except it also updates the base register with the effective address after moving the byte to memory.

Table B-88: Gas Syntax for `stb`

| Instruction | Description |
|----------------------------------|--|
| <code>stb Rs, disp(Rb)</code> | $\text{mem8}[\text{disp} + \text{Rb}] := \text{Rd}$ s and b are register numbers in the range 0..31. disp is a 16-bit signed constant. $\text{mem8}[\text{--}]$ is the byte at the memory address specified by $\text{disp} + \text{Rb}$. If Rb is RO, then this instruction substitutes the value zero for RO. |
| <code>stbu Rs, disp(Rb)</code> | $\text{mem8}[\text{disp} + \text{Rb}] := \text{Rd}$ $\text{Rs} := \text{disp} + \text{Rs}$ s and b are register numbers in the range 0..31. disp is a 16-bit signed constant. $\text{mem8}[\text{--}]$ is the byte at the memory address specified by $\text{disp} + \text{Rb}$. If Rb is RO, then this instruction substitutes the value zero for RO. |
| <code>stbx Rs, Rb, Rx</code> | $\text{mem8}[\text{Rb} + \text{Rx}] := \text{Rd}$ s , b , and x are register numbers in the range 0..31. If Rb is RO, then this instruction uses zero as the value for Rs . |
| <code>stbux Rd, Rs, Rx</code> | $\text{mem8}[\text{Rb} + \text{Rx}] := \text{Rd}$ $\text{Rb} := \text{Rb} + \text{Rx}$ s , b , and x are register numbers in the range 0..31. If Rb is RO, then this instruction uses zero as the value for Rs . |

B.55 sth, sthu, sthux, sthx

The `sth` (store halfword) instruction stores the LO 16 bits of a register into memory at an address specified by the register plus displacement addressing mode.

The `sthu` (store halfword with update) works in a similar manner except that it also updates the source register with the effective address of the halfword in memory.

The `sthx` (store halfword, indexed) stores the halfword held in the LO 16 bits of the source register into the memory location specified by the register plus register indexed addressing mode.

The `sthux` (store halfword indexed, with update) is just like `sthx` except it also updates the base register with the effective address after moving the halfword to memory.

Table B-89: Gas Syntax for `sth`

| Instruction | Description |
|----------------------------------|--|
| <code>sth Rs, disp(Rb)</code> | $\text{mem16}[\text{disp} + \text{Rb}] := \text{Rd}$ <i>s</i> and <i>b</i> are register numbers in the range 0..31. <i>disp</i> is a 16-bit signed constant. $\text{mem16}[\text{--}]$ is the halfword at the memory address specified by <i>disp</i> + <i>Rb</i> . If <i>Rb</i> is <code>RO</code> , then this instruction substitutes the value zero for <code>RO</code> . |
| <code>sthu Rs, disp(Rb)</code> | $\text{mem16}[\text{disp} + \text{Rb}] := \text{Rd}$ $\text{Rs} := \text{disp} + \text{Rs}$ <i>s</i> and <i>b</i> are register numbers in the range 0..31. <i>disp</i> is a 16-bit signed constant. $\text{mem16}[\text{--}]$ is the halfword at the memory address specified by <i>disp</i> + <i>Rb</i> . If <i>Rb</i> is <code>RO</code> , then this instruction substitutes the value zero for <code>RO</code> . |
| <code>sthx Rs, Rb, Rx</code> | $\text{mem16}[\text{Rb} + \text{Rx}] := \text{Rd}$ <i>s</i> , <i>b</i> , and <i>x</i> are register numbers in the range 0..31. $\text{mem16}[\text{--}]$ is the halfword at the memory address specified by <i>Rb</i> + <i>Rx</i> . If <i>Rb</i> is <code>RO</code> , then this instruction uses zero as the value for <i>Rs</i> . |
| <code>sthux Rd, Rs, Rx</code> | $\text{mem16}[\text{Rb} + \text{Rx}] := \text{Rd}$ $\text{Rb} := \text{Rb} + \text{Rx}$ <i>s</i> , <i>b</i> , and <i>x</i> are register numbers in the range 0..31. $\text{mem16}[\text{--}]$ is the halfword at the memory address specified by <i>Rb</i> + <i>Rx</i> . If <i>Rb</i> is <code>RO</code> , then this instruction uses zero as the value for <i>Rs</i> . |

B.56 `stmw`

The `stmw` (store multiple words) writes the values in a group of registers to a contiguous block of memory. This instruction has two operands: a starting destination register and a register plus displacement effective memory address. This instruction stores all the register values from the destination register through `R31` starting at the specified memory location. This instruction is quite useful for saving a batch of scratch-pad registers or for quickly moving blocks of memory around.

Table B-90: Gas Syntax for `stmw`

| Instruction | Description |
|----------------------------------|--|
| <code>stmw Rd, disp(Rs)</code> | $\text{mem32}[\text{disp} + \text{Rs}] \dots := \text{Rd}.. \text{R31}$ <i>d</i> and <i>s</i> are register numbers in the range 0..31 and <i>s</i> must be less than <i>d</i> . <i>disp</i> is a 16-bit signed constant. $\text{mem32}[\text{--}] \dots$ represents <i>n</i> consecutive 32-bit words in memory, where $n = 32 - d$. |

B.57 stw, stwu, stwux, stwx

The `stw` (store word) instruction stores a register's value into memory at an address specified by the register plus displacement addressing mode.

The `stwu` (store word with update) works in a similar manner except that it also updates the base register with the effective address of the word in memory.

The `stwx` (store word, indexed) stores the word held in the source register into the memory location specified by the register plus register indexed addressing mode.

The `stwux` (store word indexed, with update) is just like `stwx` except it also updates the base register with the effective address after moving the halfword to memory.

Table B-91: Gas Syntax for `sth`

| Instruction | Description |
|----------------------------------|--|
| <code>stw Rs, disp(Rb)</code> | $\text{mem32}[\text{disp} + \text{Rb}] := \text{Rd}$ s and b are register numbers in the range 0..31. disp is a 16-bit signed constant. $\text{mem32}[\text{--}]$ is the word at the memory address specified by $\text{disp} + \text{Rb}$. If Rb is RO, then this instruction substitutes the value zero for RO. |
| <code>stwu Rs, disp(Rb)</code> | $\text{mem32}[\text{disp} + \text{Rb}] := \text{Rd}$ $\text{Rs} := \text{disp} + \text{Rs}$ s and b are register numbers in the range 0..31. disp is a 16-bit signed constant. $\text{mem32}[\text{--}]$ is the word at the memory address specified by $\text{disp} + \text{Rb}$. If Rb is RO, then this instruction substitutes the value zero for RO. |
| <code>stwx Rs, Rb, Rx</code> | $\text{mem32}[\text{Rb} + \text{Rx}] := \text{Rd}$ s , b , and x are register numbers in the range 0..31. $\text{mem32}[\text{--}]$ is the word at the memory address specified by $\text{Rb} + \text{Rx}$. If Rb is RO, then this instruction uses zero as the value for Rs . |
| <code>stwux Rd, Rs, Rx</code> | $\text{mem32}[\text{Rb} + \text{Rx}] := \text{Rd}$ $\text{Rb} := \text{Rb} + \text{Rx}$ s , b , and x are register numbers in the range 0..31. $\text{mem32}[\text{--}]$ is the word at the memory address specified by $\text{Rb} + \text{Rx}$. If Rb is RO, then this instruction uses zero as the value for Rs . |

B.58 sub, sub., subo, subo.

The `sub` instruction (subtract) requires three register operands—a destination register and two source registers. This instruction computes the difference of the values in the two source registers and places the difference into the destination register. This instruction is actually a synonym for the `subf` instruction (with the register positions swapped); see `subf` for details.

Table B-92: Gas Syntax for sub

| Instruction | Description |
|---|--|
| sub <i>Rd</i> , <i>Rs1</i> , <i>Rs2</i> | $Rd := Rs1 - Rs2$ <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| sub. <i>Rd</i> , <i>Rs1</i> , <i>Rs2</i> | $Rd := Rs1 - Rs2$ CRO reflects the result of the difference. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| subo <i>Rd</i> , <i>Rs1</i> , <i>Rs2</i> | $Rd := Rs1 - Rs2$ The overflow and summary overflow bits in XER are set if a signed overflow occurs. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| subo. <i>Rd</i> , <i>Rs1</i> , <i>Rs2</i> | $Rd := Rs1 - Rs2$ CRO reflects the result of the difference. The overflow and summary overflow bits in XER are set if a signed overflow occurs. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |

Table B-93: CRO Settings for sub. and subo.

| Flag | Setting |
|------|--|
| LT | Set if the sum (signed) is less than zero. |
| GT | Set if the sum (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | The summary overflow bit from the XER is copied to this field after computing the sum. |

Table B-94: XER Settings for subo and subo.

| Flag | Setting |
|------|---|
| OV | Set if a signed overflow occurred during the execution of the instruction. |
| SO | Set if the SO bit was previously set, or if a signed overflow occurred during the execution of the instruction. |
| CA | Unaffected. |

B.59 subf, subf., subfo, subfo.

The `subf` instruction (subtract from) requires three register operands—a destination register and two source registers. This instruction computes the difference of the values in the two source registers and places the difference into the destination register. Note that this instruction subtracts the value of the first source operand from the second source operand. Assemblers create the `sub` instruction by reversing the two source operands in the actual opcode.

Table B-95: Gas Syntax for `subf`

| Instruction | Description |
|----------------------------------|--|
| <code>subf Rd, Rs1, Rs2</code> | $Rd := Rs2 - Rs1$ <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| <code>subf. Rd, Rs1, Rs2</code> | $Rd := Rs2 - Rs1$ CRO reflects the result of the difference. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| <code>subfo Rd, Rs1, Rs2</code> | $Rd := Rs2 - Rs1$ The overflow and summary overflow bits in XER are set if a signed overflow occurs. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| <code>subfo. Rd, Rs1, Rs2</code> | $Rd := Rs2 - Rs1$ CRO reflects the result of the difference. The overflow and summary overflow bits in XER are set if a signed overflow occurs. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |

Table B-96: CRO Settings for `subf.` and `subfo.`

| Flag | Setting |
|------|--|
| LT | Set if the sum (signed) is less than zero. |
| GT | Set if the sum (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | The summary overflow bit from the XER is copied to this field after computing the sum. |

Table B-97: XER Settings for `subfo` and `subfo.`

| Flag | Setting |
|------|---|
| OV | Set if a signed overflow occurred during the execution of the instruction. |
| SO | Set if the SO bit was previously set, or if a signed overflow occurred during the execution of the instruction. |
| CA | Unaffected |

B.60 `subi`

The `subi` instruction (subtract immediate) subtracts a constant from the contents of a source register and stores the difference into a destination register. The constant is limited to a signed 16-bit value (which the instruction sign extends to 32 bits prior to use). This instruction does not affect any flags or the overflow bit.

Table B-98: Gas Syntax for `subi`

| Instruction | Description |
|-------------------------------------|---|
| <code>subi Rd, Rs1, constant</code> | $Rd := Rs1 - constant$ <i>d</i> and <i>s1</i> are register numbers in the range 0..31. This instruction is a synonym for <code>addi Rd, Rs, -constant</code> . |

B.61 `subis`

The `subis` instruction (subtract immediate, shifted) shifts a 16-bit constant to the left 16 bits, subtracts this from the value in a source register, and then stores the difference into a destination register.

Table B-99: Gas Syntax for `subis`

| Instruction | Description |
|-------------------------------------|---|
| <code>subis Rd, Rs, constant</code> | $Rd := Rs - (constant \ll 16)$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. This instruction is a synonym for <code>addis Rd, Rs, -constant</code> . |

B.62 `xor, xor.`

The `xor` instruction requires three register operands—a destination register and two source registers. This instruction computes the logical (bitwise) XOR of the two source values and places the result in the destination register.

Table B-100: Gas Syntax for `xor`

| Instruction | Description |
|--------------------------------|--|
| <code>xor Rd, Rs1, Rs2</code> | $Rd := Rs1 \text{ XOR } Rs2$ <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |
| <code>xor. Rd, Rs1, Rs2</code> | $Rd := Rs1 \text{ XOR } Rs2$ CRO reflects the result of the operation. <i>d</i> , <i>s1</i> , and <i>s2</i> are register numbers in the range 0..31. |

Table B-101: CRO Settings for `xor.`

| Flag | Setting |
|------|--|
| LT | Set if the result (signed) is less than zero. |
| GT | Set if the result (signed) is greater than zero. |
| Zero | Set if the sum is zero. |
| SO | Unaffected |

B.63 xori

The `xori` (exclusive-or immediate) instruction requires two register operands and a 16-bit constant. This instruction logically exclusive-ORs the constant with the value held in the source register, and then places the result in the destination register.

Table B-102: Gas Syntax for `xori`

| Instruction | Description |
|-------------------------------------|--|
| <code>xoris Rd, Rs, constant</code> | $Rd := Rs \text{ XOR } constant$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. |

B.64 xoris

The `xoris` (exclusive-or immediate, shifted) instruction requires two register operands and a 16-bit constant. This instruction shifts the constant to the left 16 bits, logically exclusive-ORs this with the value held in the source register, and then places the result in the destination register.

Table B-103: Gas Syntax for `xoris`

| Instruction | Description |
|-------------------------------------|---|
| <code>xoris Rd, Rs, constant</code> | $Rd := Rs \text{ XOR } (constant \ll 16)$ <i>d</i> and <i>s</i> are register numbers in the range 0..31. |