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.

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

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
`add Rd, Rs1, Rs2`	`Rd := Rs1 + Rs2` *d*, *s1*, and *s2* are register numbers in the range 0..31.
`add. Rd, Rs1, Rs2`	`Rd := Rs1 + Rs2` CR0 reflects the result of the sum. *d*, *s1*, and *s2* are register numbers in the range 0..31.
`addo Rd, Rs1, Rs2`	`Rd := Rs1 + Rs2` The overflow and summary overflow bits in XER are set if a signed overflow occurs. *d*, *s1*, and *s2* are register numbers in the range 0..31.
`addo. Rd, Rs1, Rs2`	`Rd := Rs1 + Rs2` CR0 reflects the result of the sum. The overflow and summary overflow bits in XER are set if a signed overflow occurs. *d*, *s1*, and *s2* are register numbers in the range 0..31.

Table B-2: CR0 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
addi R*d*, R*s1*, *constant*	R*d* := R*s1* + *constant* *d* and *s1* 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 addi instruction uses the value zero rather than the value held in the R0 register.

Table B-5: Gas Syntax for addis

Instruction	Description
addis R*d*, R*s1*, *constant*	R*d* := R*s1* + (*constant* << 16) *d* and *s1* 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 Rd, Rs1, Rs2`	`Rd := Rs1 AND Rs2` *d*, *s1*, and *s2* are register numbers in the range 0..31.
`and. Rd, Rs1, Rs2`	`Rd := Rs1 AND Rs2` CR0 reflects the result of the operation. *d*, *s1*, and *s2* are register numbers in the range 0..31.

Table B-7: CR0 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 Rd, Rs1, Rs2`	`Rd := Rs1 AND (NOT Rs2)` *d*, *s1*, and *s2* are register numbers in the range 0..31.
`andc. Rd, Rs1, Rs2`	`Rd := Rs1 AND (NOT Rs2)` CR0 reflects the result of the operation. *d*, *s1*, and *s2* are register numbers in the range 0..31.

Table B-9: CR0 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
`andi Rd, Rs, constant`	`Rd := Rs AND constant` *d* and *s* 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
`andis Rd, Rs, constant`	`Rd := Rs AND (constant << 16)` *d* and *s* 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. *`displacement`* 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.
	`displacement` 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 conditional 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 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.
`bns target`	Branch if not summary overflow. If the SO 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.

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
cmp Rs1, Rs2	CR0 := Rs1 CMP Rs2 s1 and s2 are register numbers in the range 0..31.

Table B-18: CR0 Settings for cmp

Flag	Setting
LT	Set if the value in Rs1 (signed) is less than Rs2.
GT	Set if the value in Rs1 (signed) is greater than Rs2.
Zero	Set values in Rs1 and Rs2 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
cmpi Rs, *constant*	CR0 := Rs CMP *constant* s is a register number in the range 0..31. *constant* is a 16-bit signed constant.

Table B-20: CR0 Settings for `cmpi`

Flag	Setting
LT	Set if the value in `Rs1` (signed) is less than *`constant`*.
GT	Set `Rs`'s value (signed) is greater than *`constant`*.
Zero	Set value in `Rs1` is equal to *`constant`*.
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 CR0 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
`divw Rd, Rs1, Rs2`	`Rd := Rs1 / Rs2` *`d`*, *`s1`*, and *`s2`* are register numbers in the range 0..31.
`divw. Rd, Rs1, Rs2`	`Rd := Rs1 / Rs2` CR0 reflects the result of the quotient. *`d`*, *`s1`*, and *`s2`* are register numbers in the range 0..31.

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

Instruction	Description
`divwo Rd, Rs1, Rs2`	`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.
`divwo. Rd, Rs1, Rs2`	`Rd := Rs1 / Rs2` CR0 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-22: CR0 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 CR0 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
`divwu Rd, Rs1, Rs2`	`Rd := Rs1 / Rs2` *d*, *s1*, and *s2* are register numbers in the range 0..31.
`divwu. Rd, Rs1, Rs2`	`Rd := Rs1 / Rs2` CR0 reflects the result of the quotient. *d*, *s1*, and *s2* are register numbers in the range 0..31.

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

Instruction	Description
`divwuo Rd, Rs1, Rs2`	`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.
`divwuo. Rd, Rs1, Rs2`	`Rd := Rs1 / Rs2` CR0 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: CR0 Settings for `divwu.` and `divwuo.`

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 `divwuo` and `divwuo.`

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
`equ Rd, Rs1, Rs2`	`Rd := Rs1 == Rs2` *d*, *s1*, and *s2* are register numbers in the range 0..31.
`equ. Rd, Rs1, Rs2`	`Rd := Rs1 == Rs2` CR0 reflects the result of the operation. *d*, *s1*, and *s2* are register numbers in the range 0..31.

Table B-28: CR0 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
`extsb Rd, Rs`	`Rd := signExtend( Rs[0..7] )` *d* and *s* are register numbers in the range 0..31.
`extsb. Rd, Rs`	`Rd := signExtend( Rs[0..7] )` *d* and *s* are register numbers in the range 0..31.

Table B-30: CR0 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 Rd, Rs`	`Rd := signExtend( Rs[0..15] )` *d* and *s* are register numbers in the range 0..31.
`extsh. Rd, Rs`	`Rd := signExtend( Rs[0..15] )` *d* and *s* are register numbers in the range 0..31.

Table B-32: CR0 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 Rd, disp(Rs)`	`Rd := constant + Rs`
	d and *s* are register numbers in the range 0..31.
	This instruction is equivalent to:
	`addi Rd, Rs, constant`

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
`lbz Rd, disp( Rs )`	`Rd := zeroExtend( mem8[disp + Rs ] )` *d* and *s* are register numbers in the range 0..31. *disp* is a 16-bit signed constant. `mem8[ -- ]` is the byte at the memory address specified by *disp* + `Rs`. If `Rs` is R0, then this instruction substitutes the value zero for R0.

`lbzu Rd, disp( Rs )`	`Rd := zeroExtend( mem8[disp + Rs ] )`
	`Rs := disp + Rs`
	d and *s* are register numbers in the range 0..31.
	disp is a 16-bit signed constant.
	If `Rs` is R0, or `Rs` and `Rd` are the same, this is an invalid instruction.

`lbzx Rd, Rs, Rx`	`Rd := zeroExtend( mem8[ Rs + Rx ] )` *d*, *s*, and *x* are register numbers in the range 0..31. If `Rs` is R0, then this instruction uses zero as the value for `Rs`.

`lbzux Rd, Rs, Rx`	`Rd := zeroExtend( mem8[ Rs + Rx ] )`
	`Rs := Rs + Rx`
	d, *s*, and *x* are register numbers in the range 0..31.
	If `Rs` is R0, or `Rs` and `Rd` 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 Rd, disp( Rs )`	`Rd := signExtend( mem16[disp + Rs ] )` *d* and *s* are register numbers in the range 0..31. *disp* is a 16-bit signed constant. `mem16[ -- ]` is the 16-bit halfword at the memory address specified by *disp* + Rs. If Rs is R0, then this instruction substitutes the value zero for R0.

`lhau Rd, disp( Rs )`	`Rd := signExtend( mem16[disp + Rs ] )`
	`Rs := disp + Rs`
	d and *s* are register numbers in the range 0..31.
	disp is a 16-bit signed constant.
	If Rs is R0, or Rs and Rd are the same, this is an invalid instruction.

`lhax Rd, Rs, Rx`	`Rd := signExtend( mem16[ Rs + Rx ] )` *d*, *s*, and *x* are register numbers in the range 0..31. If Rs is R0, then this instruction uses zero as the value for Rs.

`lhaux Rd, Rs, Rx`	`Rd := signExtend( mem16[ Rs + Rx ] )`
	`Rs := Rs + Rx`
	d, *s*, and *x* are register numbers in the range 0..31.
	If Rs is R0, or Rs and Rd 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 Rd, disp( Rs )`	`Rd := zeroExtend( mem16[disp + Rs ] )` *d* and *s* are register numbers in the range 0..31. *disp* is a 16-bit signed constant. `mem16[ -- ]` is the 16-bit halfword at the memory address specified by *disp* + Rs. If Rs is R0, then this instruction substitutes the value zero for R0.

`lhzu Rd, disp( Rs )`	`Rd := zeroExtend( mem16[disp + Rs ] )`
	`Rs := disp + Rs`
	d and *s* are register numbers in the range 0..31.
	disp is a 16-bit signed constant.
	If Rs is R0, or Rs and Rd are the same, this is an invalid instruction.

`lhzx Rd, Rs, Rx`	`Rd := zeroExtend( mem16[ Rs + Rx ] )` *d*, *s*, and *x* are register numbers in the range 0..31. If Rs is R0, then this instruction uses zero as the value for Rs.

`lhzux Rd, Rs, Rx`	`Rd := zeroExtend( mem16[ Rs + Rx ] )`
	`Rs := Rs + Rx`
	d, *s*, and *x* are register numbers in the range 0..31.
	If Rs is R0, or Rs and Rd 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 Rd, constant`	`Rd := constant`
	d is a register number in the range 0..31.
	This instruction is equivalent to:
	`addi Rd, 0, constant`

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 Rd, constant`	`Rd := (constant << 16)`
	d is a register number in the range 0..31.
	This instruction is a synonym for:
	`addis Rd, 0, constant`

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
lmw R*d*, *disp*(Rs)	R*d*..R31 := mem32[*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. 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 lszux (load word and zero, indexed, with update) is just like lszx 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 `lwz`

Instruction	Description
`lwz Rd, disp( Rs )`	`Rd := mem32[disp + Rs ]` *d* and *s* are register numbers in the range 0..31. *disp* is a 16-bit signed constant. `mem32[ -- ]` is the 32-bit word at the memory address specified by *disp* + Rs. If Rs is R0, then this instruction substitutes the value zero for R0.

`lwzu Rd, disp( Rs )`	`Rd := mem32[disp + Rs]`
	`Rs := disp + Rs`
	d and *s* are register numbers in the range 0..31.
	disp is a 16-bit signed constant.
	If Rs is R0, or Rs and Rd are the same, this is an invalid instruction.

`lszx Rd, Rs, Rx`	`Rd := mem32[ Rs + Rx ]` *d*, *s*, and *x* are register numbers in the range 0..31. If Rs is R0, then this instruction uses zero as the value for Rs.

`lwzux Rd, Rs, Rx`	`Rd := mem32[ Rs + Rx ]`
	`Rs := Rs + Rx`
	d, *s*, and *x* are register numbers in the range 0..31.
	If Rs is R0, or Rs and Rd 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
`mcrf CRd, CRs`	`CRd := CRs` *d* and *s* 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
`mcrxr CRd`	`CRd := XER[0..3]` *d* is a condition-code register number in the range 0..7.

Table B-43: CR*d* 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 mfcr

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

Table B-45: Gas Syntax for `mfcr`

Instruction	Description
`mfcr Rd`	`Rd := CR[0..7]` *d* is a general-purpose register number in the range 0..31.

B.29 mfctr

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

Table B-46: Gas Syntax for `mfctr`

Instruction	Description
`mfctr Rd`	`Rd := COUNT` *d* 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
`mflr Rd`	`Rd := LINK` *d* 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
`mr Rd, Rs`	`Rd := Rs` *d* and *s* 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
`mtcrf bitmap, Rd`	`CRn := Rd[n*4..n*4+3]`, but only if `bitmap[n] == 1` *d* is a general-purpose register number in the range 0..31. *bitmap* 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
`mtctr Rd`	`COUNT := Rd` *d* 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
`mtlr Rd`	`LINK := Rd` *d* 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
`mtxer Rd`	`XER := Rd` *d* 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
`mulhw Rd, Rs1, Rs2`	`Rd := HO32( Rs1 × Rs2 )` (signed) *d, s1,* and *s2* are register numbers in the range 0..31.
`mulhw. Rd, Rs1, Rs2`	`Rd := HO32( Rs1 × Rs2 )` (signed) *d, s1,* and *s2* are register numbers in the range 0..31. This form updates CR0 (see Table B-54).

Table B-54: CR0 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 Rd, Rs1, Rs2`	`Rd := HO32( Rs1 × Rs2 )` (unsigned) *d*, *s1*, and *s2* are register numbers in the range 0..31.
`mulhwu. Rd, Rs1, Rs2`	`Rd := HO32( Rs1 × Rs2 )` (unsigned) *d*, *s1*, and *s2* are register numbers in the range 0..31. This form updates CR0 (see Table B-56).

Table B-56: CR0 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 Rd, Rs, constant`	`Rd := Rs × constant` *d* and *s* are register numbers in the range 0..31. *constant* 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
`mullw Rd, Rs1, Rs2`	`Rd := Rs1 × Rs2` (LO 32 bits) *d*, *s1*, and *s2* are register numbers in the range 0..31.
`mullwo Rd, Rs1, Rs2`	`Rd := Rs1 × Rs2` (LO 32 bits) *d*, *s1*, and *s2* are register numbers in the range 0..31. This form updates XER.
`mullw. Rd, Rs1, Rs2`	`Rd := Rs1 × Rs2` (LO 32 bits) *d*, *s1*, and *s2* are register numbers in the range 0..31. This form updates CR0.
`mullwo. Rd, Rs1, Rs2`	`Rd := Rs1 × Rs2` (LO 32 bits) *d*, *s1*, and *s2* are register numbers in the range 0..31. This form updates XER and CR0.

Table B-59: CR0 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 Rd, Rs1, Rs2	Rd := Rs1 NAND Rs2 d, s1, and s2 are register numbers in the range 0..31.
nand. Rd, Rs1, Rs2	Rd := Rs1 NAND Rs2 CR0 reflects the result of the operation. d, s1, and s2 are register numbers in the range 0..31.

Table B-62: CR0 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 Rd, Rs	Rd := -Rs d and s are register numbers in the range 0..31.
neg. Rd, Rs	Rd := -Rs CR0 reflects the result of the negation. d and s are register numbers in the range 0..31.
nego Rd, Rs	Rd := -Rs 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). d and s are register numbers in the range 0..31.
nego. Rd, Rs	Rd := Rs CR0 reflects the result of the sum. The overflow and summary overflow bits in XER are set if a signed overflow occurs. d and s are register numbers in the range 0..31.

Table B-64: CR0 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: CR0 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
`nor Rd, Rs1, Rs2`	`Rd := Rs1 NOR Rs2` *d*, *s1*, and *s2* are register numbers in the range 0..31.
`nor. Rd, Rs1, Rs2`	`Rd := Rs1 NOR Rs2` CR0 reflects the result of the operation. *d*, *s1*, and *s2* are register numbers in the range 0..31.

Table B-67: CR0 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
or Rd, Rs1, Rs2	Rd := Rs1 OR Rs2 d, s1, and s2 are register numbers in the range 0..31.
or. Rd, Rs1, Rs2	Rd := Rs1 OR Rs2 CR0 reflects the result of the operation. d, s1, and s2 are register numbers in the range 0..31.

Table B-69: CR0 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
orc Rd, Rs1, Rs2	Rd := Rs1 OR (NOT Rs2) d, s1, and s2 are register numbers in the range 0..31.
orc. Rd, Rs1, Rs2	Rd := Rs1 OR (NOT Rs2) CR0 reflects the result of the operation. d, s1, and s2 are register numbers in the range 0..31.

Table B-71: CR0 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: CR0 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
`oris Rd, Rs, constant`	`Rd := Rs OR constant` *d* and *s* 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
`oris Rd, Rs, constant`	`Rd := Rs OR (constant << 16)` *d* and *s* 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
`rlwimi Rd,Rs,n,mb,me`	`Rd := (Rd AND mask0( mb..me )) OR ((Rd ROL n) AND 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. `mask0( a..b)` is a set of zero bits in positions `a..b` and ones everywhere else. `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.
`rlwimi. Rd,Rs,n,mb,me`	`Rd := (Rd AND mask0( mb..me )) OR ((Rd ROL n) AND mask1( mb..me))` CR0 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. `mask0( a..b)` is a set of zero bits in positions `a..b` and ones everywhere else. `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: CR0 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
`rlwinm Rd,Rs,n,mb,me`	`Rd := (Rd ROL n) AND mask(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. `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.
`rlwinm. Rd,Rs,n,mb,me`	`Rd := (Rd ROL n) AND mask(mb..me)` CR0 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. `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-77: CR0 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 `rlwnm`

Instruction	Description
`rlwnm Rd,Rs,Rc,mb,me`	`Rd := (Rd ROL Rc) AND mask(mb..me)` *mb* and *me* specify the beginning and ending bit positions for the mask. `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.
`rlwnm. Rd,Rs,Rc,mb,me`	`Rd := (Rd ROL Rc) AND mask(mb..me)` CR0 reflects the result of the operation. *mb* and *me* specify the beginning and ending bit positions for the mask. `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: CR0 Settings for `rlwnm`.

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
`slw Rd,Rs,Rc`	`Rd := (Rs SHL Rc)` *d*, *s*, and *c* are register numbers in the range 0..31.
`slw. Rd,Rs,Rc`	`Rd := (Rs SHL Rc)` CR0 reflects the result of the operation. *d*, *s*, and *c* are register numbers in the range 0..31.

Table B-81: CR0 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: CR0 Settings for `slw.` (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
`sraw Rd,Rs,Rc`	`Rd := (Rs SHR Rc)` (signed) *d*, *s*, and *c* are register numbers in the range 0..31.
`sraw. Rd,Rs,Rc`	`Rd := (Rs SHR Rc)` (signed) CR0 reflects the result of the operation. *d*, *s*, and *c* are register numbers in the range 0..31.

Table B-83: CR0 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
`srawi Rd,Rs,constant`	`Rd := (Rs SHR constant)` (signed) *constant* is the number of bits to shift, in the range 0..31. *d* and *s* are register numbers in the range 0..31.
`srawi. Rd,Rs,constant`	`Rd := (Rs SHR constant)` (signed) CR0 reflects the result of the operation. *constant* is the number of bits to shift, in the range 0..31. *d* and *s* are register numbers in the range 0..31.

Table B-85: CR0 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
`srw Rd,Rs,Rc`	`Rd := (Rs SHL Rc)` *d*, *s*, and *c* are register numbers in the range 0..31.
`srw. Rd,Rs,Rc`	`Rd := (Rs SHL Rc)` CR0 reflects the result of the operation. *d*, *s*, and *c* are register numbers in the range 0..31.

Table B-87: CR0 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
stb Rs, *disp*(Rb)	mem8[*disp* + Rb] := R*d* *s* and *b* are register numbers in the range 0..31. *disp* is a 16-bit signed constant. mem8[--] is the byte at the memory address specified by *disp* + R*b*. If R*b* is R0, then this instruction substitutes the value zero for R0.
stbu Rs, *disp*(Rb)	mem8[*disp* + Rb] := R*d* Rs := *disp* + Rs *s* and *b* are register numbers in the range 0..31. *disp* is a 16-bit signed constant. mem8[--] is the byte at the memory address specified by *disp* + R*b*. If R*b* is R0, then this instruction substitutes the value zero for R0.
stbx Rs, R*b*, R*x*	mem8[R*b* + R*x*] := R*d* *s*, *b*, and *x* are register numbers in the range 0..31. If R*b* is R0, then this instruction uses zero as the value for Rs.
stbux R*d*, Rs, R*x*	mem8[R*b* + R*x*] := R*d* R*b* := R*b* + R*x* *s*, *b*, and *x* are register numbers in the range 0..31. If R*b* is R0, 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
sth Rs, *disp*(Rb)	mem16[*disp* + Rb] := R*d* *s* and *b* are register numbers in the range 0..31. *disp* is a 16-bit signed constant. mem16[--] is the halfword at the memory address specified by *disp* + R*b*. If R*b* is R0, then this instruction substitutes the value zero for R0.
sthu Rs, *disp*(Rb)	mem16[*disp* + Rb] := R*d* Rs := *disp* + Rs *s* and *b* are register numbers in the range 0..31. *disp* is a 16-bit signed constant. mem16[--] is the halfword at the memory address specified by *disp* + R*b*. If R*b* is R0, then this instruction substitutes the value zero for R0.
sthx Rs, R*b*, R*x*	mem16[Rb + Rx] := R*d* *s*, *b*, and *x* are register numbers in the range 0..31. mem16[--] is the halfword at the memory address specified by R*b* + R*x*. If R*b* is R0, then this instruction uses zero as the value for Rs.
sthux R*d*, Rs, R*x*	mem16[R*b* + R*x*] := R*d* R*b* := R*b* + R*x* *s*, *b*, and *x* are register numbers in the range 0..31. mem16[--] is the halfword at the memory address specified by R*b* + R*x*. If R*b* is R0, then this instruction uses zero as the value for Rs.

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
stmw R*d*, *disp*(Rs)	mem32[*disp* + Rs]... := R*d*..R31 *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. mem32[--]... represents *n* 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
stw Rs, *disp*(Rb)	mem32[*disp* + Rb] := Rd *s* and *b* are register numbers in the range 0..31. *disp* is a 16-bit signed constant. mem32[--] is the word at the memory address specified by *disp* + Rb. If Rb is R0, then this instruction substitutes the value zero for R0.
stwu Rs, *disp*(Rb)	mem32[*disp* + Rb] := Rd Rs := *disp* + Rs *s* and *b* are register numbers in the range 0..31. *disp* is a 16-bit signed constant. mem32[--] is the word at the memory address specified by *disp* + Rb. If Rb is R0, then this instruction substitutes the value zero for R0.
stwx Rs, Rb, Rx	mem32[Rb + Rx] := Rd *s*, *b*, and *x* are register numbers in the range 0..31. mem32[--] is the word at the memory address specified by Rb + Rx. If Rb is R0, then this instruction uses zero as the value for Rs.
stwux Rd, Rs, Rx	mem32[Rb + Rx] := Rd Rb := Rb + Rx *s*, *b*, and *x* are register numbers in the range 0..31. mem23[--] is the word at the memory address specified by Rb + Rx. If Rb is R0, 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 Rd, Rs1, Rs2	Rd := Rs1 - Rs2 d, s1, and s2 are register numbers in the range 0..31.
sub. Rd, Rs1, Rs2	Rd := Rs1 - Rs2 CR0 reflects the result of the difference. d, s1, and s2 are register numbers in the range 0..31.
subo Rd, Rs1, Rs2	Rd := Rs1 - Rs2 The overflow and summary overflow bits in XER are set if a signed overflow occurs. d, s1, and s2 are register numbers in the range 0..31.
subo. Rd, Rs1, Rs2	Rd := Rs1 - Rs2 CR0 reflects the result of the difference. The overflow and summary overflow bits in XER are set if a signed overflow occurs. d, s1, and s2 are register numbers in the range 0..31.

Table B-93: CR0 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
`subf Rd, Rs1, Rs2`	`Rd := Rs2 - Rs1` *d*, *s1*, and *s2* are register numbers in the range 0..31.
`subf. Rd, Rs1, Rs2`	`Rd := Rs2 - Rs1` CR0 reflects the result of the difference. *d*, *s1*, and *s2* are register numbers in the range 0..31.
`subfo Rd, Rs1, Rs2`	`Rd := Rs2 - Rs1` The overflow and summary overflow bits in XER are set if a signed overflow occurs. *d*, *s1*, and *s2* are register numbers in the range 0..31.
`subfo. Rd, Rs1, Rs2`	`Rd := Rs2 - Rs1` CR0 reflects the result of the difference. The overflow and summary overflow bits in XER are set if a signed overflow occurs. *d*, *s1*, and *s2* are register numbers in the range 0..31.

Table B-96: CR0 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
`subi Rd, Rs1, constant`	`Rd := Rs1 - constant` *d* and *s1* are register numbers in the range 0..31. This instruction is a synonym for `addi Rd, Rs, -constant`.

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
`subis Rd, Rs, constant`	`Rd := Rs - (constant << 16)` *d* and *s* are register numbers in the range 0..31. This instruction is a synonym for `addis Rd, Rs, -constant`.

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
`xor Rd, Rs1, Rs2`	`Rd := Rs1 XOR Rs2` *d*, *s1*, and *s2* are register numbers in the range 0..31.
`xor. Rd, Rs1, Rs2`	`Rd := Rs1 XOR Rs2` CR0 reflects the result of the operation. *d*, *s1*, and *s2* are register numbers in the range 0..31.

Table B-101: CR0 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
xoris R*d*, Rs, *constant*	R*d* := Rs XOR *constant* *d* and *s* 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
xoris R*d*, Rs, *constant*	R*d* := Rs XOR (*constant* << 16) *d* and *s* are register numbers in the range 0..31.