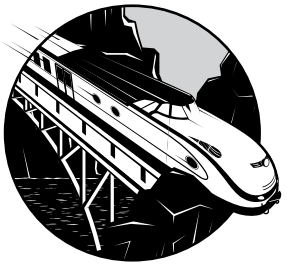
B

POWERPC ASSEMBLY FOR THE HLL PROGRAMMER



A basic understanding of PowerPC assembly language will enable you to read the PowerPC output produced by compilers on machines like the older Power Macintosh or for game consoles such as the Sony PlayStation or the Xbox. Thus, this appendix provides an overview of the following topics:

- The PowerPC machine architecture
- PowerPC assembly language
- The memory addressing modes of the PowerPC CPU
- The syntax used by the PowerPC Gas assembler
- How to use constants and declare data in assembly language programs

In addition, the resources at *www.writegreatcode.com* describe a minimal PowerPC instruction set that you'll need to examine compiler output.

B.1 Assembly Syntaxes

As Chapter 3 explained, there are significant syntax differences in the code generated by various assemblers for the 80x86. PowerPC assemblers, by contrast, use a much more uniform syntax, so you should have no trouble reading PowerPC assembly listings produced for other assemblers (such as the Code Warrior assembler) if you learn the GNU Gas syntax used in this book.

B.2 Basic PowerPC Architecture

The IBM/Motorola PowerPC CPU family is classified as a *Von Neumann machine*. Von Neumann computer systems contain three main building blocks: the *central processing unit (CPU)*, *memory*, and *input/output (I/O) devices*. These three components are connected together via the *system bus* (consisting of the address, data, and control buses). Figure 3-1 showed this relationship.

The CPU communicates with memory and I/O devices by placing a numeric value on the *address bus* to select one of the memory locations or I/O device port locations, each of which has a unique binary numeric address. Then the CPU, I/O, and memory devices pass data among themselves by placing the data on the *data bus*. The *control bus* provides signals that determine the direction of the data transfer (to/from memory and to/from an I/O device).

The registers are the most prominent feature within the CPU. The PowerPC CPU registers are categorized as general-purpose integer registers, floating-point registers, special-purpose application-accessible registers, and special-purpose kernel-mode registers. Special-purpose kernel-mode registers are intended for writing operating systems, debuggers, and other system-level tools. That topic is well beyond the scope of this book, so they will not be discussed further.

B.2.1 General-Purpose Integer Registers

The PowerPC CPUs provide 32 general-purpose integer registers for application use. Most compilers refer to these registers as R0 through R31. On older PowerPC processors (circa 2020), these registers are 32 bits wide. On higher-end PowerPC processors, they are 64 bits wide.

B.2.2 General-Purpose Floating-Point Registers

The PowerPC processors also provide 32 64-bit floating-point registers. Assemblers and compilers generally refer to these registers as F0 through F31 (or FPR0 through FPR31). These registers can hold single- or double-precision floating-point values.

B.2.3 User-Mode-Accessible Special-Purpose Registers

The user-mode-accessible special-purpose registers include the condition code register(s), the floating-point status and control register, the XER register, the LINK register (LR), the COUNT register (CTR), and the time base registers (TBRs).

B.2.3.1 Condition Code Registers

The condition code register is 32 bits wide, but it is actually a collection of eight 4-bit registers (CR0 through CR7) that hold the status of a previous computation (such as the result of a comparison). The PowerPC uses CR0 to hold the condition codes after an integer operation, and CR1 to hold the condition codes after a floating-point operation. Programs typically use the remaining condition code registers to save the status of some operation while other operations take place.

The individual bits in the CR0 condition code register are as follows:

The LT bit (CR0: bit 0) Set if an operation produces a negative result. Also indicates a "less than" condition.

The GT bit (CR0: bit 1) Set if the result is positive (and nonzero) after an operation. Also indicates a "greater than" condition.

The zero bit (CR0: bit 2) This bit is set when the result of an operation is `0`. This also indicates the "equality" condition.

The summary overflow bit (CR0: bit 3) This indicates a signed integer overflow during a chain of operations (see the XER register description in Section B.2.3.3).

The individual bits in the CR1 condition code register hold the following values:

- Floating-point exception bit (CR1: bit 0)
- Floating-point enable exception bit (CR1: bit 1)
- Floating-point invalid exception bit (CR1: bit 2)
- Floating-point overflow exception bit (CR1: bit 3)

Table B-1 describes how the PowerPC sets the CR*n* bits after a comparison instruction.

Table B-1: CR*n* Field Bit Settings for Comparisons

CR*n* bit	Meaning	Description
0	Less than (integer or floating-point)	For integer registers, this bit is set if one register is less than another (or a small immediate *constant*). Unsigned and signed comparisons are possible using different instructions. For floating-point registers, this bit is set if the value in one floating-point register is less than the value in another after a floating-point comparison (which is always a signed comparison).

(continued)

Table B-1: CR*n* Field Bit Settings for Comparisons (continued)

CR*n* bit	Meaning	Description
1	Greater than (integer or floating-point)	For integer registers, this bit is set if one register is greater than another (or a small immediate *constant*). Unsigned and signed comparisons are possible using different instructions. For floating-point registers, this bit is set if the value in one floating-point register is greater than the value in another after a floating-point comparison (which is always a signed comparison).
2	Equal (integer or floating-point)	For integer registers, this bit is set if one register is equal to another (or a small immediate *constant*). Unsigned and signed comparisons are the same when comparing for equality. For floating-point registers, this bit is set if the value in one floating-point register is equal to the value in another after a floating-point comparison.
3	Summary overflow (integer) Not A Number, NaN (floating-point)	After an integer operation, this bit indicates whether an overflow has occurred. This bit is sticky insofar as you can only clear it, once set, by explicitly clearing the SO bit in the XER register. After a floating-point operation, this bit indicates whether one of the two floating-point operands is NaN.

B.2.3.2 Floating-Point Status and Control Register

The floating-point status and control register is a 32-bit register containing 24 status bits and 8 control bits. The status bits appear in bit positions 0 through 23, and the control bits in bit positions 24 through 31. The PowerPC CPU updates the status bits at the completion of each floating-point instruction; the program is responsible for initializing and manipulating the control bits.

Most of the status bits in the floating-point status and control register are *sticky*—that is, once a bit is set, it remains set until explicitly cleared by software. This allows the CPU to execute a sequence of floating-point operations and then test for an invalid result, rather than having to test after each instruction. This reduces the number of tests an application must do and, therefore, reduces the size and increases the performance of the application.

The exact nature of the floating-point status bits is not important here (the PowerPC code in this book rarely checks the floating-point status), so we'll skip a complete discussion of it. For more details, consult the PowerPC documentation available from IBM (*www.ibm.com*).

B.2.3.3 XER Register

The PowerPC XER register collects several disparate values that don't have a home elsewhere. The low-order (LO) 3 bits maintain the summary overflow (bit 0), overflow (bit 1), and carry (bit 2) conditions. The high-order (HO) 8 bits contain a byte count for some PowerPC string operations.

The overflow and summary overflow bits are set by instructions that produce a signed integer result that cannot be represented in 32 or 64 bits (depending on the instruction). The summary overflow bit is sticky. The overflow bit, on the other hand, simply reflects the status (overflow/no overflow) of the last arithmetic operation; in particular, if an arithmetic operation does not produce an overflow, then that operation clears the overflow bit.

The carry flag is set whenever an arithmetic instruction produces an unsigned integer result that cannot be held in 32 or 64 bits (depending on the instruction); this bit also holds the bits shifted out of a register operand in a shift left or right operation.

B.2.3.4 The LINK Register

The PowerPC LINK register holds a *return address* after the execution of a *branch and link* (`bl`) instruction. That is, executing `bl` leaves the address of the instruction following the branch in the LINK register. PowerPC applications use this register to implement returns from subroutine operations as well as to compute program-counter relative addresses for various operations. The PowerPC can also use the LINK register for indirect jumps (such as for implementing `switch` statements).

B.2.3.5 The COUNT Register

The COUNT register (also called *CTR*) has two purposes: it's used as a loop control register and to hold the target address for an indirect jump. Most compilers use the COUNT register for the latter purpose, as shown throughout the code in this book.

B.2.3.6 The Time Base Registers

The Time Base Lower (TBL) and Time Base Upper (TBU) registers are read-only in user mode. Applications can use these two registers (which actually concatenate to form a single 64-bit register) to compute the execution time of an instruction sequence. However, as few compilers consider the values in these registers, this book doesn't discuss them.

B.3 Literal Constants

Like most assemblers, Gas supports literal numeric, character, and string constants. This section describes their syntax.

B.3.1 Binary Literal Constants

Binary literal constants in Gas begin with the special `0b` prefix followed by one or more binary digits (`0` or `1`). Examples:

```
0b1011
0b10101111
0b0011111100011001
0b1011001010010101
```

B.3.2 Decimal Literal Constants

Decimal literal constants in Gas take the standard form—a sequence of one or more decimal digits without any special prefix or suffix. Examples:

```
123
1209345
```

B.3.3 Hexadecimal Literal Constants

Hexadecimal literal constants in Gas consist of a string of hexadecimal digits (`0..9`, `a..f`, or `A..F`) with a `0x` prefix. Examples:

```
0x1AB0
0x1234ABCD
0xdead
```

B.3.4 Character and String Literal Constants

Character literal constants in Gas consist of an apostrophe followed by a single character. Examples:

```
'a
''
'!
```

String literal constants in Gas consist of a sequence of zero or more characters surrounded by quotes. They use the same syntax as C strings. You use the `\` escape sequence to embed special characters in a Gas string. Examples:

```
"Hello World"
"" -- The empty string
"He said \"Hello\" to them"
"\"" -- string containing a single quote character
```

B.3.5 Floating-Point Literal Constants

Floating-point literal constants in assembly language typically take the same form you'll find in HLLs—a sequence of digits, possibly containing a decimal point, optionally followed by a signed exponent. Examples:

```
3.14159
2.71e+2
1.0e-5
5e1
```

B.4 Manifest (Symbolic) Constants in Assembly Language

Almost every assembler provides a mechanism for declaring symbolic (named) constants. Gas uses the `.equ` ("equate") statement to define a symbolic constant in the source file. This statement uses the following syntax:

```
.equ        symbolName, value
```

Here are some examples within a Gas source file:

```
.equ        false, 0
.equ        true, 1
.equ        anIntConst, 12345
```

B.5 PowerPC Addressing Modes

PowerPC instructions can access three types of operands: register operands, immediate constants, and memory operands.

B.5.1 PowerPC Register Access

Gas allows assembly programmers and compiler writers to access the PowerPC general-purpose integer registers by name or number: R0 through R31.

Floating-point instructions access the floating-point registers by their name (F0 through F31). Note that floating-point registers are legal only as floating-point instruction operands (just as integer registers are accessible only within integer instructions).

B.5.2 The Immediate Addressing Mode

Many integer instructions allow a programmer to specify an immediate constant as a source operand. However, as all PowerPC instructions are exactly 32 bits in size, a single instruction cannot load a 32-bit (or larger) constant into a PowerPC register. The PowerPC's instruction set does support immediate constants that are 16 bits in size (or smaller). The PowerPC encodes those constants into the opcode and sign-extends their values to 32 bits (or 64 bits) prior to using them.

For immediate values outside the range –32,768 through +32,767, the PowerPC requires that you load the constant into a register using a couple of instructions and then use the value in that register. The most obvious downside to this is that the code is larger and slower, but another problem is that you must dedicate a register to hold the immediate value. Fortunately, the PowerPC has 32 general-purpose registers available, so using a register for this purpose isn't quite as costly as on a CPU with fewer registers (like the 80x86).

B.5.3 PowerPC Memory Addressing Modes

The PowerPC CPU is a *load/store* architecture, meaning that it can only access (data) memory using load and store instructions. All other instructions operate on registers (or small immediate constants). With a load/store architecture, for example, you cannot directly add the contents of some memory location to a register value—you must first load the memory data into a register and then add that register to the destination register's value.

RISC CPUs generally eschew complex addressing modes, instead relying upon sequences of machine instructions using simple addressing modes to achieve the same effect. The PowerPC, true to its RISC heritage, supports only three memory addressing modes. One of those is a special addressing mode used only by the load string and store string instructions. So, for all practical purposes, the PowerPC supports only two memory addressing modes: *register plus displacement* and *register plus register* (base plus index).

B.5.3.1 Register Plus Displacement Addressing Mode

The PowerPC register plus displacement addressing mode adds a signed 16-bit displacement value, sign-extended to 32 bits, with the value from a general-purpose integer register to compute the effective memory address. The Gas syntax for this addressing mode is as follows:

```
displacementValue( Rn )
```

where *displacementValue* is a signed 16-bit expression and *Rn* represents one of the PowerPC's 32-bit general-purpose integer registers (R0 through R31). R0 is a special case in this addressing mode, however. If you specify R0, the PowerPC CPU substitutes `0` in place of the value in the R0 register. This provides an *absolute* or *displacement-only* addressing mode that accesses memory locations 0 through 32,767 (and also the final 32KB at the end of the address space).

The `lbz` (load byte with zero extension) instruction is a typical load instruction that uses the register plus displacement addressing mode. It fetches a byte from memory, zero-extends it to 32 bits (64 bits on the 64-bit variants of the PowerPC), and then copies the result into a destination register. For example, this particular instruction loads the LO byte of R3 with the byte found in memory at the address held in R5 plus 4. It zeros out the HO bytes of R3:

```
lbz R3, 4(R5)
```

Most load and store instructions (like `lbz`) on the PowerPC support a special *update* form. When you're using the register plus displacement addressing mode, these instructions work just like the standard load instructions except that they update the base address register with the final effective address. That is, they add the displacement to the base register's

value after loading the value from memory. The `lbzu` instruction is a good example of this form:

```
lbzu R3, 4(R5)
```

This instruction not only copies the value from memory location [R5 + 4][1] into R3, but also adds 4 to R5. Note that you can't specify R0 as a base register when using the update form (remember, the PowerPC substitutes the `0` for R0, and you can't store a value into a constant).

B.5.3.2 Register Plus Register (Indexed) Addressing Mode

The PowerPC also supports an indexed addressing mode that uses one general-purpose register to hold a base address and a second general-purpose register to hold an index from that base address. This addressing mode is specified as part of the instruction mnemonic. For example, to use the indexed addressing mode with the `lbz` instruction, you'd use the `lbzx` mnemonic. Instructions using this addressing mode typically have three operands: a destination operation (for loads) or a source operand (for store operations), a base register (*Rb*), and an index register (*Rx*). The `lbzx` instruction, for example, uses the following syntax:

```
lbzx Rd, Rb, Rx
```

This example loads R3 with the zero-extended byte found at the memory address [R5 + R6]:

```
lbzx R3, R5, R6
```

There's also an update form of the indexed addressing mode (such as `lbzux`). This form updates the base register with the sum of the base and index registers after computing the effective memory address. The index register's value is unaffected by the update form of the instruction.

B.6 Declaring Data in Assembly Language

The PowerPC CPU provides only a few low-level machine data types on which individual machine instructions can operate:

- Bytes that hold arbitrary 8-bit values
- Words that hold arbitrary 16-bit values (*halfwords* in PowerPC terminology)
- Double words that hold arbitrary 32-bit values (*words* in PowerPC terminology)

1. The brackets ([]) denote indirection. That is, [R5 + 4] represents the memory at the address specified by the contents of R5 plus 4.

- Quad words that hold 64-bit values (*double words* in PowerPC terminology)
- Single-precision floating-point values (32-bit single floating-point values)
- Double-precision, 64-bit, floating-point values

NOTE *Although the standard PowerPC terminology is* byte, halfword, word, *and* double word *for 8-, 16-, 32-, and 64-bit integer values, outside of this appendix this book uses the x86 terminology to avoid confusion with the 80x86 code.*

Gas uses the `.byte` directive in a `.data` section to declare a byte variable, like so:

```
variableName: .byte 0
```

Gas doesn't provide an explicit form for creating uninitialized variables, so you just supply a `0` operand for them. Here is an actual byte variable declaration in Gas:

```
IntializedByte: .byte   5
```

Gas also does not provide an explicit directive for declaring an array of byte objects, but you can use the `.rept`/`.endr` directives to create multiple copies of the `.byte` directive as follows:

```
variableName:
        .rept   sizeOfBlock
        .byte   0
        .endr
```

You can also supply a comma-delimited list of values to initialize the array with different values.

Here are a couple of array declaration examples in Gas:

```
            .section    .data ; Variables go in this section
InitializedArray0:            ; Creates an array with elements 5,5,5,5
            .rept       4
            .byte       5
            .endr

InitializedArray1:
            .byte       0,1,2,3,4,5
```

For 16-bit objects, Gas uses the `.int` directive. Other than the size of the object these directives declare, their use is identical to the byte declarations:

```
                    .section    .data
GasWordVar:         .int    0

; Create an array of four words, all initialized to 0:
```

```
GasWordArray:
                    .rept   4
                    .int    0
                    .endr

; Create an array of 16-bit words, initialized with
; the values 0, 1, 2, 3, and 4:

GasWordArray2:      .int    0,1,2,3,4
```

For 32-bit objects, Gas uses the `.long` directive:

```
                    .section    .data
GasDWordVar:        .long   0

; Create an array with four double-word values
; initialized to 0:

GasDWordArray:
                    .rept   4
                    .long   0
                    .endr

; Create an array of double words initialized with
; the values 0, 1, 2, 3, 4:

GasDWordArray2:     .long   0,1,2,3,4
```

For floating-point values, Gas uses the `.single` and `.double` directives to reserve storage for an IEEE-format[2] floating-point value (32 or 64 bits, respectively). Because the PowerPC CPU does not support immediate floating-point constants, if you need to reference a floating-point constant from a machine instruction, you'll need to place that constant in a memory variable and access the memory variable instead. Here are some examples:

```
                    .section    .data
GasSingleVar:       .single  0.0
GasDoubleVar:       .double  1.0

; Create an array with four single-precision values
; initialized to 2.0:

GasSingleArray:
                    .rept   4
                    .single 2.0
                    .endr

; Create an array of double-precision values initialized with
; the values 0.0, 1.1, 2.2, 3.3, and 4.4:

GasDWordArray2:     .double 0.0,1.1,2.2,3.3,4.4
```

2. IEEE is the Institute of Electrical and Electronics Engineers.

B.7 Specifying Operand Sizes in Assembly Language

PowerPC instructions generally operate only on 32-bit or 64-bit data. Unlike CISC processors, individual PowerPC instructions don't operate on various data types. The `add` instruction, for example, operates only on 32-bit values (except on 64-bit implementations of the PowerPC, where it operates on 64-bit values when in 64-bit mode). Generally, this isn't a problem. If two PowerPC registers contain 8-bit values, you'll get the same result by adding those two 32-bit registers together that you'd get if they were 8-bit registers, if you consider only the LO 8 bits of the sum.

Memory accesses, however, are a different matter. When reading and (especially) writing data in memory, it's important that the CPU access only the desired data size. Therefore, the PowerPC provides some size-specific load and store instructions that specify byte, 16-bit halfword, and 32-bit word sizes.

B.8 The Minimal Instruction Set

Although the PowerPC CPU family supports hundreds of instructions, few compilers actually use all of them. This is because many instructions have become obsolete over time as newer instructions have emerged. Some instructions, such as PowerPC's AltiVec instructions, simply don't correspond to functions you'd normally perform in an HLL. As a result, compilers rarely generate these types of machine instructions, which generally appear only in handwritten assembly language programs. Fortunately, this means you don't need to learn the entire PowerPC instruction set in order to study compiler output, but only the handful that compilers actually emit.

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 sections group instructions together that differ only by these suffixes.

B.9 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-2: 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-3: CR0 Settings for add. and addo.

Flag	Setting
LT	Set if the sum (signed) is less than 0.
GT	Set if the sum (signed) is greater than 0.
Zero	Set if the sum is 0.
SO	The summary overflow bit from the XER is copied to this field after computing the sum.

Table B-4: 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.10 addi

The addi (add immediate) instruction 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, addi uses the value 0 rather than the value held in the R0 register. In this case, addi acts as a "load immediate with sign extension" instruction (because adding an immediate constant with 0 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'll also discover that there's no "subtract immediate" instruction, even though assemblers like Gas support that mnemonic. Gas (and other PowerPC assemblers) compiles a `subi` instruction into an `addi` instruction after negating the immediate operand.

Table B-5: Gas Syntax for `addi`

Instruction	Description
`addi Rd, Rs, constant`	*Rd* := *Rs*+ *constant* *d* and *s* are register numbers in the range `0..31`.

B.11 addis

The `addis` (add immediate, shifted) instruction 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, `addi` uses the value `0` rather than the value held in the R0 register.

Table B-6: Gas Syntax for `addis`

Instruction	Description
`addis Rd, Rs, constant`	*Rd* := *Rs* + (*constant* << 16) *d* and *s* are register numbers in the range `0..31`.

B.12 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-7: 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-8: CR0 Settings for `and.`

Flag	Setting
LT	Set if the result (signed) is less than 0.
GT	Set if the result (signed) is greater than 0.
Zero	Set if the result is 0.
SO	Unaffected.

B.13 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-9: 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-10: CR0 Settings for `andc.`

Flag	Setting
LT	Set if the result (signed) is less than 0.
GT	Set if the result (signed) is greater than 0.
Zero	Set if the result is 0.
SO	Unaffected.

B.14 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-11: 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.15 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-12: 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.16 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 may 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, and compilers like GCC typically use the synonyms rather than the numeric forms. In this section, we'll discuss these "simplified branch mnemonics." If you encounter weird forms of the branch instructions, you may want to consult *PowerPC Microprocessor Family: The Programmer's Reference Guide* (see "For More Information" on page 50) 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 we won't consider that form here.

B.16.1 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-13: 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.16.2 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-14: 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.16.3 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-15: 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.16.4 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 allows you to test the condition bits found in any of the eight PowerPC condition code registers (CR0 through 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 them here. For details on the other forms, see the PowerPC programmer's reference manual.

The conditional branches support only 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. This generally isn't much of a problem, as conditional branches do not transfer control over great distances in typical programs.

Table B-16: 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.
`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.16.5 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 following syntax.

Table B-17: Indirect Conditional Branches

Indirect branch		Description
LINK	COUNT	
`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.16.6 Other Branch Forms

The PowerPC provides a bewildering array of branch instructions. We don't use many of those other forms in this book, so there's no need to consider them here. See the *PowerPC Microprocessor Family: The Programmer's Reference Guide* (*https://www.cebix.net/downloads/bebox/PRG.pdf*) for more details on the available forms of the branch instructions.

B.17 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, `cmp` assumes that you wish 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-18: 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-19: 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 if values in *Rs1* and *Rs2* are equal.
SO	Copied from the SO bit in the XER register.

B.18 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 wish to use CR0 to hold the result, though it's possible to specify a different condition code register as the target for the comparison operation.

Table B-20: 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-21: CR0 Settings for `cmpi`

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

B.19 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.20 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 through 65,535. See `cmpi` for more details.

B.21 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 (for example, a division by zero).

Table B-22: 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.
`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-23: CR0 Settings for `divw.` and `divwo.`

Flag	Setting
LT	Set if the quotient (signed) is less than 0.
GT	Set if the quotient (signed) is greater than 0.
Zero	Set if the quotient is 0.
SO	The summary overflow bit from the XER is copied to this field after computing the sum.

Table B-24: 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.22 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 (for example, a division by zero).

Table B-25: 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`.
`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-26: CR0 Settings for `divwu.` and `divwuo.`

Flag	Setting
LT	Set if the quotient is less than `0`.
GT	Set if the quotient is greater than `0`.
Zero	Set if the quotient is `0`.
SO	The summary overflow bit from the XER is copied to this field after computing the sum.

Table B-27: 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.23 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. EQU does a bit-by-bit comparison of two 32-bit values and stores a `1` in the corresponding destination bit position if the two source bit values are equal, and stores a `0` in the destination bit position if the two source bits are not equal.

Table B-28: 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-29: CR0 Settings for `equ.`

Flag	Setting
LT	Set if the result (signed) is less than `0`.
GT	Set if the result (signed) is greater than `0`.
Zero	Set if the result is `0`.
SO	Unaffected.

B.24 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-30: 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-31: CR0 Settings for `extsb.`

Flag	Setting
LT	Set if the result (signed) is less than 0.
GT	Set if the result (signed) is greater than 0.
Zero	Set if the result is 0.
SO	Unaffected.

B.25 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-32: 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-33: CR0 Settings for `extsh.`

Flag	Setting
LT	Set if the result (signed) is less than 0.
GT	Set if the result (signed) is greater than 0.
Zero	Set if the result is 0.
SO	Unaffected.

B.26 la

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

Table B-34: 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.27 lbz, lbzu, lbzux, lbzx

The lbz (load byte and zero) instruction fetches a byte from memory at an address specified by the displacement plus register 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-35: 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 0 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 0 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.28 lha, lhau, lhax, lhaux

The lha (load halfword, algebraic) instruction fetches a 16-bit word from memory at an address specified by the displacement plus register 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-36: 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 0 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 0 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.29 lhz, lhzu, lhzx, lhzux

The lhz (load halfword and zero) instruction fetches a 16-bit word from memory at an address specified by the displacement plus register 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-37: 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 0 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 0 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.30 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-38: 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.31 lis

The lis (load immediate, shifted) instruction 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-39: 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.32 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 displacement plus register 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-40: Gas Syntax for lmw

Instruction	Description
lmw *Rd*, *disp*(*Rs*)	*Rd*..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.33 lwz, lwzu, lwzx, lwzux

The lwz (load word and zero) instruction fetches a 32-bit word from memory at an address specified by the displacement plus register 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-41: 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 0 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 0 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.34 mcrf

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

Table B-42: 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.35 mcrxr

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

Table B-43: 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-44: 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-45: XER Settings for `mcrxr.`

Flag	Setting
SO	0
OV	0
CA	0

B.36 mfcr

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

Table B-46: 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.37 mfctr

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

Table B-47: Gas Syntax for `mfctr`

Instruction	Description
`mfctr` *Rd*	*Rd* := COUNT *d* is a general-purpose register number in the range 0..31.

B.38 mflr

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

Table B-48: Gas Syntax for `mflr`

Instruction	Description
`mflr Rd`	*Rd* := LINK *d* is a general-purpose register number in the range `0..31`.

B.39 mr

The `mr` (move register) instruction 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. See the `or` instruction for more details.

Table B-49: Gas Syntax for `mr`

Instruction	Description
`mr Rd, Rs`	*Rd* := *Rs* *d* and *s* are register numbers in the range `0..31`.

B.40 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 `0`, then the corresponding bits in the condition code field are unaffected by this instruction.

Table B-50: 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.41 mtctr

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

Table B-51: Gas Syntax for mtctr

Instruction	Description
mtctr *Rd*	COUNT := *Rd* *d* is a general-purpose register number in the range 0..31.

B.42 mtlr

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

Table B-52: Gas Syntax for mtlr

Instruction	Description
mtlr *Rd*	LINK := *Rd* *d* is a general-purpose register number in the range 0..31.

B.43 mtxer

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

Table B-53: Gas Syntax for mtxer

Instruction	Description
mtxer *Rd*	XER := *Rd* *d* is a general-purpose register number in the range 0..31.

B.44 mulhw, mulhw.

The mulhw (multiply high word) instruction produces the HO 32 bits of a 32 × 32 multiplication 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-54: 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-55).

Table B-55: CR0 Settings for `mulhw.`

Flag	Setting
LT	Set if the signed result is less than 0.
GT	Set if the signed result is greater than 0.
Zero	Set if the result is equal to 0.
SO	Copied from the SO bit in the XER register.

B.45 mulhwu, mulhwu.

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

Table B-56: 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-55).

Table B-57: CR0 Settings for `mulhwu.`

Flag	Setting
LT	Set if the signed result is less than 0.
GT	Set if the signed result is greater than 0.
Zero	Set if the result is equal to 0.
SO	Copied from the SO bit in the XER register.

B.46 mulli

The `mulli` (multiply low word, immediate) 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. 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-58: 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.47 mullw, mullw., mullwo, mullwo.

The `mullw` (multiply low word) instruction computes 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-59: 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-60: CR0 Settings for `mullw.` and `mullwo.`

Flag	Setting
LT	Set if the signed result is less than `0`.
GT	Set if the signed result is greater than `0`.
Zero	Set if the result is equal to `0`.
SO	Copied from the SO bit in the XER register.

Table B-61: XER Settings for `mullwo` and `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.48 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-62: 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-63: CR0 Settings for nand.

Flag	Setting
LT	Set if the result (signed) is less than 0.
GT	Set if the result (signed) is greater than 0.
Zero	Set if the result is 0.
SO	Unaffected.

B.49 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-64: 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-65: CR0 Settings for neg. and nego.

Flag	Setting
LT	Set if the result is less than 0.
GT	Set if the result is greater than 0.
Zero	Set if the result is 0.
SO	The summary overflow bit from the XER is copied to this field after computing the sum.

Table B-66: 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.50 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-67: 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-68: CR0 Settings for nor.

Flag	Setting
LT	Set if the result (signed) is less than 0.
GT	Set if the result (signed) is greater than 0.
Zero	Set if the result is 0.
SO	Unaffected.

B.51 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-69: 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-70: CR0 Settings for or.

Flag	Setting
LT	Set if the result (signed) is less than 0.
GT	Set if the result (signed) is greater than 0.
Zero	Set if the result is 0.
SO	Unaffected.

B.52 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-71: 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-72: CR0 Settings for `orc.`

Flag	Setting
LT	Set if the result (signed) is less than 0.
GT	Set if the result (signed) is greater than 0.
Zero	Set if the result is 0.
SO	Unaffected.

B.53 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-73: 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.54 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-74: 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.55 rlwimi, rlwimi.

The `rlwimi` (rotate left word immediate, then mask insert) instruction 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* and *me* (the second and third immediate operands) from this result and inserts them into the destination register (without affecting the bits outside the range *mb* through *me* in the destination register).

Table B-75: 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 0 bits in positions a..b and 1s everywhere else. mask1(a..b) is a set of 1 bits in positions a..b and 0s 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 0 bits in positions a..b and 1s everywhere else. mask1(a..b) is a set of 1 bits in positions a..b and 0s everywhere else. *d* and *s* are register numbers in the range 0..31.

Table B-76: CR0 Settings for `rlwimi.`

Flag	Setting
LT	Set if the result (signed) is less than 0.
GT	Set if the result (signed) is greater than 0.
Zero	Set if the result is 0.
SO	Unaffected.

B.56 rlwinm, rlwinm.

The `rlwinm` (rotate left word immediate, then AND with mask) instruction 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` and `me` (the second and third immediate operands) from this result and stores the result into the destination register (with 0s in bit positions outside the mask range).

Table B-77: 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 1 bits in positions a..b and 0s 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 1 bits in positions a..b and 0s everywhere else. *d* and *s* are register numbers in the range 0..31.

Table B-78: CR0 Settings for `rlwinm.`

Flag	Setting
LT	Set if the result (signed) is less than 0.
GT	Set if the result (signed) is greater than 0.
Zero	Set if the result is 0.
SO	Unaffected.

B.57 rlwnm, rlwnm.

The `rlwnm` (rotate left word then AND with mask) instruction 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` and `me` (the second and third immediate operands) from this result and stores the result into the destination register (with 0s in bit positions outside the mask range).

Table B-79: 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 1 bits in positions a..b and 0s everywhere else. *d*, *s*, and *c* are register numbers in the range 0..31.

(continued)

Table B-79: Gas Syntax for `rlwnm` (continued)

Instruction	Description
`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 `1` bits in positions `a..b` and `0`s everywhere else. *d* and *s* are register numbers in the range `0..31`.

Table B-80: CR0 Settings for `rlwnm.`

Flag	Setting
LT	Set if the result (signed) is less than `0`.
GT	Set if the result (signed) is greater than `0`.
Zero	Set if the result is `0`.
SO	Unaffected.

B.58 slw, slw.

The `slw` (shift left word) instruction requires three register operands—a destination register, a source register, and 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-81: 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-82: CR0 Settings for `slw.`

Flag	Setting
LT	Set if the result (signed) is less than `0`.
GT	Set if the result (signed) is greater than `0`.
Zero	Set if the result is `0`.
SO	Unaffected.

B.59 sraw, sraw.

The sraw (shift right, arithmetic, word) instruction requires three register operands—a destination register, a source register, and 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-83: 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-84: CR0 Settings for sraw.

Flag	Setting
LT	Set if the result (signed) is less than 0.
GT	Set if the result (signed) is greater than 0.
Zero	Set if the result is 0.
SO	Unaffected.

B.60 srawi, srawi.

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-85: 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-86: CR0 Settings for `srawi.`

Flag	Setting
LT	Set if the result (signed) is less than 0.
GT	Set if the result (signed) is greater than 0.
Zero	Set if the result is 0.
SO	Unaffected.

B.61 srw, srw.

The `srw` (shift right word) instruction requires three register operands—a destination register, a source register, and 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 0s into the unoccupied HO bit positions. Bits shifted out of the LO bit position are lost.

Table B-87: 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-88: CR0 Settings for `srw.`

Flag	Setting
LT	Set if the result (signed) is less than 0.
GT	Set if the result (signed) is greater than 0.
Zero	Set if the result is 0.
SO	Unaffected.

B.62 stb, stbu, stbux, stbx

The `stb` (store byte) instruction stores the LO byte of a register into memory at an address specified by the displacement plus register 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-89: Gas Syntax for stb

Instruction	Description
stb *Rs*, *disp*(*Rb*)	mem8[*disp* + *Rb*] := *Rs* *s* and *b* are register numbers in the range 0..31. *disp* is a 16-bit signed *constant*. mem8[*disp* + *Rb*] is the byte at the memory address specified by *disp* + *Rb*. If *Rb* is R0, then this instruction substitutes the value 0 for R0.
stbu *Rs*, *disp*(*Rb*)	mem8[*disp* + *Rb*] := *Rs* *Rb* := *disp* + *Rb* *s* and *b* are register numbers in the range 0..31. *disp* is a 16-bit signed *constant*. mem8[*disp* + *Rb*] is the byte at the memory address specified by *disp* + *Rb*. If *Rb* is R0, then this instruction substitutes the value 0 for R0.
stbx *Rs*, *Rb*, *Rx*	mem8[*Rb* + *Rx*] := *Rs* *s*, *b*, and *x* are register numbers in the range 0..31. If *Rb* is R0, then this instruction uses 0 as the value for *Rs*.
stbux *Rs*, *Rb*, *Rx*	mem8[*Rb* + *Rx*] := *Rs* *Rb* := *Rb* + *Rx* *s*, *b*, and *x* are register numbers in the range 0..31. If *Rb* is R0, then this instruction uses 0 as the value for *Rb*.

B.63 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 displacement plus register 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-90: Gas Syntax for sth

Instruction	Description
sth *Rs*, *disp*(*Rb*)	mem16[*disp* + *Rb*] := *Rs* *s* and *b* are register numbers in the range 0..31. *disp* is a 16-bit signed constant. mem16[*disp* + *Rb*] is the halfword at the memory address specified by *disp* + *Rb*. If *Rb* is R0, then this instruction substitutes the value 0 for R0.
sthu *Rs*, *disp*(*Rb*)	mem16[*disp* + *Rb*] := *Rs* *Rb* := *disp* + *Rb* *s* and *b* are register numbers in the range 0..31. *disp* is a 16-bit signed constant. mem16[*disp* + *Rb*] is the halfword at the memory address specified by *disp* + *Rb*. If *Rb* is R0, then this instruction substitutes the value 0 for R0.
sthx *Rs*, *Rb*, *Rx*	mem16[*Rb* + *Rx*] := *Rs* *s*, *b*, and *x* are register numbers in the range 0..31. mem16[*Rb* + *Rx*] is the halfword at the memory address specified by *Rb* + *Rx*. If *Rb* is R0, then this instruction uses 0 as the value for *Rb*.
sthux *Rs*, *Rs*, *Rx*	mem16[*Rb* + *Rx*] := *Rs* *Rb* := *Rb* + *Rx* *s*, *b*, and *x* are register numbers in the range 0..31. mem16[*Rb* + *Rx*] is the halfword at the memory address specified by *Rb* + *Rx*. If *Rb* is R0, then this instruction uses 0 as the value for *Rs*.

B.64 stmw

The stmw (store multiple words) instruction 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 displacement plus register effective memory address. stmw 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-91: Gas Syntax for stmw

Instruction	Description
stmw *Rs*, *disp*(*Rd*)	mem32[*disp* + *Rd*]... := *Rs*..R31 *d* and *s* are register numbers in the range 0..31, and *d* must be less than *s*. *disp* is a 16-bit signed constant. mem32[*disp* + *Rd*]... represents *n* consecutive 32-bit words in memory, where *n* = 32 – *s*.

B.65 stw, stwu, stwux, stwx

The stw (store word) instruction stores a register's value into memory at an address specified by the displacement plus register 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-92: Gas Syntax for sth

Instruction	Description
stw *Rs*, *disp*(*Rb*)	mem32[*disp* + *Rb*] := *Rs* *s* and *b* are register numbers in the range 0..31. *disp* is a 16-bit signed constant. mem32[*disp* + *Rb*] is the word at the memory address specified by *disp* + *Rb*. If *Rb* is R0, then this instruction substitutes the value 0 for R0.
stwu *Rs*, *disp*(*Rb*)	mem32[*disp* + *Rb*] := *Rs* *Rs* := *disp* + *Rs* *s* and *b* are register numbers in the range 0..31. *disp* is a 16-bit signed constant. mem32[*disp* + *Rb*] is the word at the memory address specified by *disp* + *Rb*. If *Rb* is R0, then this instruction substitutes the value 0 for R0.
stwx *Rs*, *Rb*, *Rx*	mem32[*Rb* + *Rx*] := *Rs* *s*, *b*, and *x* are register numbers in the range 0..31. mem32[*Rb* + *Rx*] is the word at the memory address specified by *Rb* + *Rx*. If *Rb* is R0, then this instruction uses 0 as the value for *Rs*.
stwux *Rs*, *Rb*, *Rx*	mem32[*Rb* + *Rx*] := *Rs* *Rb* := *Rb* + *Rx* *s*, *b*, and *x* are register numbers in the range 0..31. mem32[*Rb* + *Rx*] is the word at the memory address specified by *Rb* + *Rx*. If *Rb* is R0, then this instruction uses 0 as the value for *Rb*.

B.66 sub, sub., subo, subo.

The sub (subtract) instruction 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 result 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-93: 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-94: CR0 Settings for sub. and subo.

Flag	Setting
LT	Set if the difference (signed) is less than 0.
GT	Set if the difference (signed) is greater than 0.
Zero	Set if the difference is 0.
SO	The summary overflow bit from the XER is copied to this field after computing the sum.

Table B-95: 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.67 subf, subf., subfo, subfo.

The subf (subtract from) instruction 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 result 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-96: 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-97: CR0 Settings for `subf.` and `subfo.`

Flag	Setting
LT	Set if the difference (signed) is less than 0.
GT	Set if the difference (signed) is greater than 0.
Zero	Set if the difference is 0.
SO	The summary overflow bit from the XER is copied to this field after computing the sum.

Table B-98: 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.68 subi

The `subi` (subtract immediate) instruction 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-99: 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.69 subis

The subis (subtract immediate, shifted) instruction 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-100: 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.70 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-101: 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-102: CR0 Settings for xor.

Flag	Setting
LT	Set if the result (signed) is less than 0.
GT	Set if the result (signed) is greater than 0.
Zero	Set if the result is 0.
SO	Unaffected.

B.71 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-103: Gas Syntax for `xori`

Instruction	Description
`xoris Rd, Rs, constant`	*Rd* := *Rs* XOR *constant* *d* and *s* are register numbers in the range `0..31`.

B.72 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-104: Gas Syntax for `xoris`

Instruction	Description
`xoris Rd, Rs, constant`	*Rd* := *Rs* XOR (*constant* << 16) *d* and *s* are register numbers in the range `0..31`.

B.73 For More Information

Blanchard, Hollis. "PowerPC Assembly." July 2, 2002. *https://ibm.co/2I1uzSm*.

IBM Knowledge Center. "Appendix F: PowerPC Instructions." n.d. *https://ibm.co/2PzVzg2*.

Motorola, Inc., and IBM. *PowerPC Microprocessor Family: The Programmer's Reference Guide*. Chicago: Motorola, Inc., 1995. *https://www.cebix.net/downloads/bebox/PRG.pdf*.