PowerPC User-Level Instruction Set Quick Reference Card

Based on a mnemonic presentation idea from Bill Karsh in his PowerPC tutorial series in MacTech magazine (http://macte.ch/luHry)

Notation

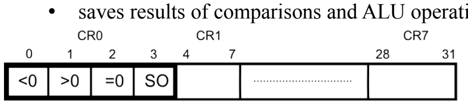
||	Concatenation of bit blocks
|	Alternation
UIMMnn	Unsigned immediate of *nn* bits (ie: UIMM16 = 16 bits)
SIMMnn	Signed immediate of nn bits (ie: SIMM26 = 26 bits)
EXT	Sign-extend to word
(rA|0)	In some instances, value "0" for register rA (meaning r0) is a special case that actually means "use the value 0".
< >	List of functional suffixes (append 0 or 1 from the list)
[ ]	List of optional suffixes (0 or more, in the order specified)

Example of multiple suffixes for an instruction:

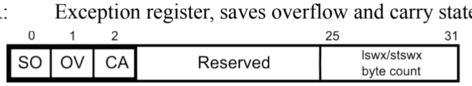
add<c | e | me | ze>[o, .]: **add**, **addc**, **addme, addze, addo, addco**, **addeo**, **addmeo**, **addzeo**, **add.**, **addc.**, **adde.**, **addme.**, **addze.**, **addo.**, **addeo.**, **addmeo.**, **addzeo**. are all valid.

Registers

- r0-r31: General-purpose integer registers
- LR: Link register, saves return address of branches that link
- CTR: Counter for auto-decrementing loops
- CR: Condition register
  - composed of 8 condition records (CR0-CR7)
  - saves results of comparisons and ALU operations



- XER: Exception register, saves overflow and carry states



Label suffixes

Assemblers general recognize several suffixes for labels and values. These suffixes provide ways to extract only parts of an operand for use in immediate values.

- VALUE**@h**: Only the high 16-bit part (bits 0-15).
- VALUE**@ha**: Like @h, but adjusted to compensate for sign extension applied by an "addi VALUE@l" on the same register.
- VALUE**@l**: Only the low 16-bit part (bits 16-31).

NOTE: Compare the following two ways to load an immediate value into a register (equivalent in result but different in spirit):

1. addis rD, 0, VALUE@ha
   addi rD, rD, VALUE@l
2. addis rD, 0, VALUE@h
   ori rD, rD, VALUE@l

With the first method, the **addi** instruction does sign extension on its 16-bit signed immediate operand. If we want to load, for instance 0x12348765, the value 0x8765 from "0x12348765**@l**" will be sign-extended to 0xFFFF8765. This will cause an off-by-one error if we add it with 0x12340000 from **addis** rD,0, 0x12348765**@h**. The **@ha** suffix verifies this condition ("negative" low 16 bits) and adjusts the high-part so that when it is added with the sign-extended low part, the result correct: 0x12348765 in our case. With the second method, using **ori** which does not sign-extend its operand, the high part requires no adjustment.

Load and store instructions

Addressing modes

The PowerPC has only two addressing modes, but combining them with the load/store instructions options yields many possibilities. The addressing modes (using **lwz** as an example) are:

- **lwz** rD, offset(rA|0) → Register-indirect with immediate offset
  → EA = (rA + offset) or (0 + offset)
  → Offset is a 16 bit signed immediate value
- **lwzx** rD, (rA|0), rB → Register-indirect with indexing
  → EA = (rA + rB) or (0 + rB)

Single loads and stores

Instruction	Operation
`lbz[u,x] rD,d(rA)`	rD ← *byte from* MEM[EA]
`lhz[u,x] rD,d(rA)`	rD ← *half-word from* MEM[EA]
`lha[u,x] rD,d(rA)`	rD ← *sign-extended half word from* MEM[EA]
`lwz[u,x] rD,d(rA)`	rD ← word from MEM[EA]
`stb[u,x] rS,d(rA)`	rS[24:31] → MEM[EA] (*store byte*)
`sth[u,x] rS,d(rA)`	rS[16:31] → MEM[EA] (*store half-word*)
`stw[u,x] rS,d(rA)`	rS → MEM[EA] (*store word*)

- **"z"** load suffix: treat as unsigned, zero-extend, right-justify.
- **"a"** load suffix: "algebraic": sign-extend to word.
- `[u]`: "update": if (rA != 0) then rA←EA after load or store. In the case of loads, condition (rD != rA) also applies (logically so).
- `[x]`: "with indexing" (see addressing modes above), use operands as in "lwzx rD, (rA|0), rB" instead of "lwz rD, d(RA)".

Multiple loads and stores

Instruction	Operation
`lmw rD,d(rA)`	n = (32 – rD); n consecutive words starting at EA are loaded into GPRs rD through r31. For example : lmw r29,0(r8) loads r29, r30 and r31 from consecutive, increasing addresses starting at EA.
`stmw rS,d(rA)`	n = (32 – rS); n consecutive words starting at EA are stored from the GPRs rS through r31. For example, if rS = 29, r29, 30 and r31 are stored at consecutive, increasing addresses starting at EA.
String loads and stores (lswi, lswx, stswi, stswx) are omitted for brevity and because they are not available on all PPCs.	

Arithmetic and logic instructions

Addition, subtraction, negation

Instruction	Operands	Operation
`add<c,e>[o,.]`	`rD,rA,rB`	rD ← rA + rB
`addi<s,c,c.>`	`rD,(rA|0),SIMM`	rD ← (rA|0) + EXT(SIMM16)
`addme[o,.]`	`rD,rA`	rD ← rA + XER[CA] - 1
`addze[o,.]`	`rD,rA`	rD ← rA + 0 + XER[CA]
`neg[o,.]`	`rD,rA`	rD ← (¬rA + 1) *(2's complement negation)*
`subf<c,e>[o,.]`	`rD,rA,rB`	rD ← rB – rA
`subfic`	`rD,rA,SIMM`	rD ← EXT(SIMM16) – rA
`subfme[o,.]`	`rD,rA`	rD ← -1 – rA + XER[CA]
`subfze[o,.]`	`rD,rA`	rD ← 0 – rA + XER[CA]

- **"i"** suffix: "immediate": second operand is 16-bit sign-extended immediate value.
- **"s"** suffix: "shifted": immediate value is logical shifted left 16 bits prior to being used.
- **"z"** suffix: replaces rB with immediate value 0 (0x00000000).
- **"m"** suffix: replaces rB with immediate value -1 (0xFFFFFFFF).
- **"e"** suffix: extended add or subtract. The value of `XER[CA]` is added to the result, enabling multi-word carry arithmetic. The value of `XER[CA]` is updated by these operations also.
- **"c"** suffix: carry updated. `XER[CA]` is updated with the operation's carry state (by default, the carry is unaffected).
- `[o]`: overflow updated. `XER[OV]` and `XER[SO]` are updated according to whether the operation overflows or not.
- `[.]`: Record result of operation in `CR0` (<0, >0, =0, SO)

NOTE: Many features have been omitted for brevity. This includes Floating-Point operations and registers, supervisor-level operations, string load and stores and ALTIVEC instructions.

Bitwise logical operations and shifts

Instruction	Operands	Operation
`and[c,.]`	`rD,rA,rB`	rD ← rA ∧ rB
`andi.`	`rD,rA,UIMM`	rD ← rA ∧ UIMM16
`andis.`	`rD,rA,UIMM`	rD ← rA ∧ (UIMM16 << 16)
`cntlzw[.]`	`rD,rA`	rD ← *number of leading zeros in* rA
`eqv[.]`	`rD,rA,rB`	rD ← ¬(rA ⊕ rB) *(would be "xnor")*
`extsb[.]`	`rD,rA`	rD ← EXT(rA[24:31]) (*sign-extend low byte of* rA)
`extsh[.]`	`rD,rA`	rD ← EXT(rA[16:31]) (*sign-extend low half-word of* rA)
`nand[.]`	`rD,rA,rB`	rD ← ¬(rA ∧ rB)
`nor[.]`	`rD,rA,rB`	rD ← ¬(rA ∨ rB)
`or[c,.]`	`rD,rA,rB`	rD ← rA ∨ rB
`ori`	`rD,rA,UIMM`	rD ← rA ∨ (UIMM16)
`oris`	`rD,rA,UIMM`	rD ← rA ∨ (UIMM16 << 16)
`slw[.]`	`rD,rA,rB`	rD ← rA << rB[26:31] (logical)
`srw[.]`	`rD,rA,rB`	rD ← rA >> rB[26:31] (logical)
`srawi[.]`	`rD,rA,UIMM`	rD ← rA >> UIMM5 (arithmetic)
`sraw[.]`	`rD,rA,rB`	rD ← rA >> rB[26:31] (arithmetic)
`xor[c,.]`	`rD,rA,rB`	rD ← rA ⊕ rB
`xori`	`rD,rA,UIMM`	rD ← rA ⊕ (UIMM16)
`xoris`	`rD,rA,UIMM`	rD ← rA ⊕ (UIMM16 << 16)

- `[c]`: Complement (invert) the value from rB prior to using it. The actual value residing in rB is unaffected.
- `[.]`: Record result of operation in CR0 (<0, >0, =0, SO)
- NOTE: on shifts, values 0-32 are valid. For arithmetic shift rights, a value of 32 fills the word with the sign bit. For logical shift lefts, a value of 32 sets the word to 0.

Multiplication

Inst.	Operands	Operation
`mulhw`	`rD,rA,rB`	rD ← rA × rB *(32 upper bits of 64-bit result)*
`mulli`	`rD,rA,SIMM`	rD ← (rA × SIMM16) *(32 lower bits of 48-bit result)*
`mullw`	`rD,rA,rB`	rD ← rA × rB *(32 lower bits of 64-bit result)*

Division

Inst.	Operands	Operation
`divw<u>[o,.]`	`rD,rA,rB`	rD ← rA ÷ rB

- "`u`" suffix: treat operands as unsigned numbers
- `[o]`: Record overflow of result
- `[.]`: Record result of operation in CR0 (<0, >0, =0, SO)

Rotate and mask

Inst.	Operands	Operation
`rlwimi[.]`	`rD,rA,UIMM,MB,ME`	rD ← rotate rA left by UIMM bits, mask and insert result in rD
`rlwinm[.]`	`rD,rA,UIMM,MB,ME`	rD ← *rotate rA left by UIMM bits and mask*
`rlwnm[.]`	`rD,rA,rB,MB,ME`	rD ← *rotate rA left by rB bits and mask*

- For all these instructions, a mask M is built by starting with a zero-word (0x00000000) and setting bits to "1" starting at bit number MB and ending at bit number ME, both inclusive. It is possible to wrap-around while generating the mask (ie: MB > ME).

Examples:

MASK(MB,ME) with MB=29 and ME=3:

0↓	1↓	2↓	3 ME	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29 MB	30↓	31↓
1	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**

MASK(MB,ME) with MB=8 and ME=14:

0	1	2	3	4	5	6	7	8 MB	9↓	10↓	11↓	12↓	13↓	14 ME	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

`rlwimi r3,r4,6,20,25 (r4 = 0x0FF0_0017, r3 = 0x12AB_CDEF)`

1. Generate mask : MASK(20,25) = `0x0000_0FC0`
2. Rotate source: tmp1 = r4 ROL 6 = `0xFC00_05C3`
3. Extract field: tmp2 = tmp1 ∧ MASK = `0x0000_05C0`
4. Mask destination: tmp3 = r3 ∧ ¬MASK = `0x12AB_C02F`
5. Insert field in destination: r3 ← tmp2 ∨ tmp3 = `0x12AB_C5EF`

The previous 5 steps as binary:

```
1. 0b0000_0000_0000_0000_0000_1111_1100_0000
2. 0b1111_1100_0000_0000_0000_0101_1100_0011
3. 0b0000_0000_0000_0000_0000_0101_1100_0000
4. 0b0001_0010_1010_1011_1100_0000_0010_1111
5. 0b0001_0010_1010_1011_1100_0101_1110_1111
```

`rlwinm r3,r4,12,20,31 (r4 = 0x5A70_00BB)`

1. Generate mask : MASK(20,31) = `0x0000_0FFF`
2. Rotate source: tmp1 = r4 ROL 12 = `0x000B_B5A7`
3. Extract field in destination: r3 ← tmp1 ∨ MASK = `0x0000_05A7`

Comparison instructions

Inst.	Operands	Operation
`cmp`	`crD,L,rA,rB`	Compare signed rA to rB
`cmpi`	`crD,L,rA,SIMM`	Compare signed rA to EXT(SIMM16)
`cmpl`	`crD,L,rA,rB`	Compare unsigned rA to rB
`cmpli`	`crD,L,rA,UIMM`	Compare unsigned rA to (0x0000 ‖ UIMM16)

- `crD` can be omitted. In that case, the assembler assumes `cr0.`
- The L field means "Long" (64-bit compare) if set to "1", or 32-bit compare if set to "0". On 32-bit PowerPC, L should always be set to "0". Because of this, a simplified mnemonic exists for all "cmp"-series instructions: "`cmpw crD, rA, rB`" is equivalent to "`cmp crD,0,rA,rB`", etc.
- For all these instructions, the result of a comparison from rA to (rB|SIMM|UIMM) is stored in the specified condition register crD. For example, cmp 3,0,rA,rB would yield `cr3` = "`100` ‖ `XER[SO]`" if rA < rB, `cr3` = "`010` ‖ `XER[SO]`" if rA > rB and `cr3` = "`001` ‖ `XER[SO]`" if rA = rB.

Condition register manipulation instructions

Inst.	Operands	Operation
`crand`	`crbD,crbA,crbB`	crbD ← crbA ∧ crbB
`crandc`	`crbD,crbA,crbB`	crbD ← crbA ∧ ¬crbB
`creqv`	`crbD,crbA,crbB`	crbD ← ¬(crbA ⊕ crbB)
`crnand`	`crbD,crbA,crbB`	crbD ← ¬(crbA ∧ crbB)
`crnor`	`crbD,crbA,crbB`	crbD ← ¬(crbA ∨ crbB)
`cror`	`crbD,crbA,crbB`	crbD ← crbA ∨ crbB
`crorc`	`crbD,crbA,crbB`	crbD ← crbA ∨ ¬crbB
`crxor`	`crbD,crbA,crbB`	crbD ← crbA ⊕ crbB
`mcrf`	`crD,crA`	crD ← crA (*move field A to field D*)
`crclr`	`crbD`	*Simplified for* `crxor crbD,crbD,crbD`
`crmove`	`crbD,crbA`	*Simplified for* `cror crbD,crbA,crbA`
`crnot`	`crbD,crbA`	*Simplified for* `crnor crbD,crbA,crbA`
`crset`	`crbD`	*Simplified for* `creqv crbD,crbD,crbD`

- For the `cr<OP>` instructions, operands `crb[A,B,D]` mean "condition register bit", with value 0-31. All of these instructions carry-out logical operations between single bits of the CR, no matter what the conventional "meanings" of the bits are (<0, >0, =0, SO).

Example:

- `cror 0,5,6` : `CR[0]` ← `CR[5]` ∨ `CR[6]` , thus `cr0[=0]` ← 1, if `cr1` had ">=" comparison result, otherwise `cr0[=0]` ← 0.

NOTE: Many features have been omitted for brevity. This includes Floating-Point operations and registers, supervisor-level operations, string load and stores and ALTIVEC instructions.

Branch instructions

The PowerPC architecture uses a very flexible branching unit to decode the several fields contained in branch instructions. We will cover the basic branch instructions and their fields, and then present tables and examples of simplified branch mnemonics.

Field names

- BI (Branch Input): which bit of the CR is used as a branch condition
- BO (Branch Options): how to treat CTR and BI to determine if branching occurs
- Target: where to branch

Branch instructions

Inst.	Operands	Operation
`b[l,a]`	`target`	Branch unconditionally
`bc[l,a]`	`BO,BI,target`	Branch conditionally
`bclr[l]`	`BO,BI`	Branch to LR conditionally
`bcctr[l]`	`BO,BI`	Branch to CTR conditionally

- `[l]`: linking: store current PC + 4 in LR, so that a "`blr`" instruction can be used to return from a function call.
- `[a]`: absolute: target is an absolute address instead of a PC-relative displacement.
    - For the `b[l,a]` instruction, target is a 26-bit signed immediate with 2 LSbs always "0" (4-bytes aligned). Maximum branch distance is [-33,554,432...33,554,428].
    - For the `bc[l,a]` instruction, target is a 16-bit signed immediate with 2 LSbs always "0" (4-bytes aligned). Maximum branch distance is [-32,768...32764].
    - In the case of non-absolute (no `[a]` option) branches, the target displacement is added to PC. A displacement of 0 is an infinite loop at the current PC. For the unconditional branch (`[a]` option), the target is still signed, but the displacement is based around 0x0000_0000.
- To obtain the value of PC, one can branch linking to the next instruction (`bl +4`). The LR will contain the PC value at that next instruction. This trick is used by compilers to access local constant pools inserted after function return instructions.
- PowerPC assemblers and linkers will always adjust relocations so that displacements and labels can be specified directly, without having to adjust the value formats to the field formats. For example, "`b +8`" will get encoded as a target of 0x000002 (stripped of the 2 LSbs) automatically in the instruction.
- BI values can be simplified with constants named `cr0` through `cr7` with values 0-7 respectively and constants named lt,gt,eq,so with values 0-3 respectively . Then, `cr4[<0]`, which is BI=16 can be written as `(cr4*4)+lt`.

BO values

BO	Branch if	Symbol
`0000y`	Decremented CTR ≠ 0 and the condition is **false**.	`dnzf`
`0001y`	Decremented CTR = 0 and the condition is **false**.	`dzf`
`001zy`	Branch if the condition is **false**.	`f`
`0100y`	Decremented CTR ≠ 0 and the condition is **true**.	`dnzt`
`0101y`	Decremented CTR = 0 and the condition is **true**.	`dzt`
`011zy`	Branch if the condition is **true**.	`t`
`1z00y`	Decremented CTR ≠ 0 (only CTR checked).	`dnz`
`1z01y`	Decremented CTR = 0 (only CTR checked).	`dz`
`1z1zz`	Branch always.	–

- Symbols (3rd column of table above): the "c" of "`bc`" and the BO field value can be omitted and replaced with one of these symbols as a suffix. For example, and assuming y=z=0, the instruction "`bc 8,5,label`" can be replaced with "`bdnzt 5,label`".
- "y" bits are "branch likely to be taken" hints if set to "1". This is ignored by many implementations. A suffix of "-" added to the instruction clears this bit (branch not likely taken). A suffix of "+" added to the instruction sets this bit (branch likely taken). Example: "`bdnzt+ 5,label`" is equivalent to "`bc 9,5,label`". Many processors of the PowerPC family ignore this hint.
- "z" bits should be zeroed as they are for future extensions.

Examples:

- `bc 8,5,label` : Branch if decremented CTR ≠ 0 and CR[5] = "1".
- `bdnzt 5,label` : same as above.
- `bdnzt (cr1*4)+gt,label` : same as above.
- `bl label`: Branch and link to label (call function, return with `blr`).
- `blr`: Branch unconditional to LR (return from function).
- `bdza label`: Branch absolute to label if decremented CTR = 0.
- `btctr lt`: Branch to CTR if `cr0[<0]` (`CR[0]`) = "1".
- `bf eq,label`: Branch to label if `cr0[=0]` (`CR[2]`) = "0".

Simplified branches (or "classic" branches)

There are simplified "branch conditional" mnemonics that emulate the classic branches of other instruction sets. These mnemonics are for instructions that do not test the CTR.

Instruction	Operands	Operation
`b<test>[l,a]`	`[crN,]target`	Branch conditionally
`b<test>lr[l]`	`[crN]`	Branch to LR conditionally
`b<test>ctr[l]`	`[crN]`	Branch to CTR conditionally

- `[crN]` is an optional CR subfield number (ie: `cr0-cr7`), on which the test will take place. If omitted, the default is `cr0`.

Simplified branches tests (using b<test> as example)

Symbol	Branch if
`beq`	Equal, or zero (`cr[=0]` = "1")
`bge`	Greater than or equal (`cr[>0]` = "1" ∨ `cr[=0]` = "1")
`bgt`	Greater than (`cr[>0]` = "1")
`ble`	Less than or equal (`cr[<0]` = "1" ∨ `cr[=0]` = "1")
`blt`	Less than (`cr[<0]` = "1")
`bne`	No equal, or not zero (`cr[=0]` = "0")
`bng`	Not greater than (equivalent to b**le**)
`bnl`	Not less than (equivalent to b**ge**)
`bns`	Not summary overflow (`cr[SO]` = "0")
`bso`	Summary overflow (`cr[SO]` = "1")

Examples:

- `bne label` : Branch to label if `cr0[=0]` = "0".
- `bsola cr2,label` : Branch absolute linking to label if `cr2[SO]` = "1".
- `bltl label` : Branch linking to label if `cr0[<0]` = "1".
- `beqctr` cr4 : Branch to CTR if `cr4[=0]` = "1".
- `bgtlrl` : Branch linking to LR if `cr0[>0]` = "1".
- `bl label`: Branch and link to label (call function, return with `blr`).
- `blr`: Branch unconditional to LR (return from function).

Special Purpose Register (SPR) Operations

Inst.	Operands	Operation
`mcrxr`	`crD`	crD ←XER[0:3] then zero XER[0:3]
`mfcr`	`rD`	rD ← CR[0:31]
`mfspr`	`rD,SPR`	rD ← SPR
`mtcrf`	`crM,rS`	CR updated with rS[crM] (*see notes below*)
`mtspr`	`SPR,rS`	SPR ← rS
`mtcr`	`rS`	*Simplified for* `mtcrf 0xFF, rS`

- `crM` is an 8 bit immediate mask (value 0x00-0xFF). The MSb means `cr0`, the LSb means `cr7`, and bits in between mean `cr1-cr6`. For example, `crM = 0xA2 = 0b1010_0010` would mean to load `cr0`, `cr2` and `cr6` from rS into the CR, and leave the other fields (`cr1`,`cr3`,`cr4`,`cr5` and `cr7`) intact.
- There are simplified `mtspr` mnemonics for several SPRs which allow the omission of the SPR number: `mtctr`, `mtlr`, `mtxer`.
- There are simplified `mfspr` mnemonics for several SPRs which allow the omission of the SPR number: mfctr, `mflr`, `mfxer`.

NOTE: Many features have been omitted for brevity. This includes Floating-Point operations and registers, supervisor-level operations, string load and stores and ALTIVEC instructions.

Trap and System Call Instructions

Inst.	Operands	Operation
`sc`	—	System call
`tw`	`TO,rA,rB`	Trap if rA <TO> rB is true
`twi`	`TO,rA,SIMM`	Trap if rA <TO> EXT(SIMM16) is true

- TO is a 5-bit field of conditions to test. If any of the conditions are met, the trap is taken.
  - TO[0] (ie: mask = 0b10000) means (a < b)
  - TO[1] (ie: mask = 0b01000) means (a > b)
  - TO[2] (ie: mask = 0b00100) means (a = b)
  - TO[3] means (a < b) with unsigned compare
  - TO[4] means (a > b) with unsigned compare

Condensed alphabetical instructions list

Instruction		Operation
`add[.]`	`rD,rA,rB`	Add
`addc[o,.]`	`rD,rA,rB`	Add, saving carry
`adde[o,.]`	`rD,rA,rB`	Add extended (adding carry)
`addi`	`rD,(rA|0),SIMM`	Add immediate
`addis`	`rD,(rA|0),SIMM`	Add immediate shifted
`addic[.]`	`rD,(rA|0),SIMM`	Add immediate shifted saving carry
`addme[o,.]`	`rD,rA`	Add to minus one, extended
`addze[o,.]`	`rD,rA`	Add to zero, extended
`and[.]`		AND
`andc[.]`		AND with complement
`andi.`	`rD,rA,UIMM`	AND with immediate
`andis.`	`rD,rA,UIMM`	AND with shifted immediate
`b[l,a]`	`target`	Branch always
`bc[l,a]`	`BO,BI,target`	Branch conditionally
`bcctr[l]`	`BO,BI`	Branch conditionally to CTR
`bclr[l]`	`BO,BI`	Branch conditionally to LR
`beq[l,a]`	`[crN,]target`	Branch on equal (or zero)
`bge[l,a]`	`[crN,]target`	Branch on greater than or equal
`bgt[l,a]`	`[crN,]target`	Branch on greater than
`ble[l,a]`	`[crN,]target`	Branch on lower than or equal
`blt[l,a]`	`[crN,]target`	Branch on lower than
`bne[l,a]`	`[crN,]target`	Branch on not equal (or non-zero)
`bng[l,a]`	`[crN,]target`	Branch on not greater than
`bnl[l,a]`	`[crN,]target`	Branch on not lower than
`bns[l,a]`	`[crN,]target`	Branch on not summary overflow
`bso[l,a]`	`[crN,]target`	Branch on summary overflow
`cmp`	`[crD,]L,rA,rB`	Compare signed
`cmpi`	`[crD,]L,rA,SIMM`	Compare signed with immediate
`cmpl`	`[crD,]L,rA,rB`	Compare unsigned
`cmpli`	`[crD,]L,rA,UIMM`	Compare unsigned with immed.
`cntlzw[.]`	`rD,rA`	Count leading zeros in word
`crand`	`crbD,crbA,crbB`	AND on CR bits
`crandc`	`crbD,crbA,crbB`	AND complemented on CR bits
`crclr`	`crbD`	Clear CR bit
`creqv`	`crbD,crbA,crbB`	EQV on CR bits
`crmove`	`crbD,crbA`	Move CR bit
`crnand`	`crbD,crbA,crbB`	NAND on CR bits
`crnor`	`crbD,crbA,crbB`	NOR on CR bits
`crnot`	`crbD,crbA`	NOT on CR bit
`cror`	`crbD,crbA,crbB`	OR on CR bits
`crorc`	`crbD,crbA,crbB`	OR complemented on CR bits
`crset`	`crbD`	Set CR bit
`crxor`	`crbD,crbA,crbB`	XOR on CR bits
`divw[o,.]`	`rD,rA,rB`	Divide word
`divwu[o,.]`	`rD,rA,rB`	Divide word unsigned
`eqv[.]`	`rD,rA,rB`	EQV (NOT (rA XOR rB)
`extsb[.]`	`rD,rA`	Sign-extend byte
`extsh[.]`	`rD,rA`	Sign-extend half-word
`lbz[u,x]`	`rD,d(rA)`	Load byte unsigned
`lha[u,x]`	`rD,d(rA)`	Load half-word and sign-extend
`lhz[u,x]`	`rD,d(rA)`	Load half-word unsigned
`lmw`	`rD,d(rA)`	Load multiple words
`lwz[u,x]`	`rD,d(rA)`	Load word
`mcrf`	`crD,crA`	Move condition register field
`mcrxr`	`crD`	Move XER[0:3] to CR field
`mfcr`	`rD`	Move from CR
`mfspr`	`rD,SPR`	Move from SPR
`mtcr`	`rS`	Move to CR
`mtcrf`	`crM,rS`	Update CR fields
`mtspr`	`SPR,rS`	Move to SPR
`mulhw`	`rD,rA,rB`	Multiply high word
`mulli`	`rD,rA,SIMM`	Multiply low immediate
`mullw`	`rD,rA,rB`	Multiply low word
`nand[.]`	`rD,rA,rB`	NAND
`neg[o,.]`	`rD,rA`	Negate (2's complement)
`nor[.]`	`rD,rA,rB`	NOR
`or[.]`	`rD,rA,rB`	OR
`orc[.]`	`rD,rA,rB`	OR with complement
`ori`	`rD,rA,UIMM`	OR with immediate
`oris`	`rD,rA,UIMM`	OR with shifted immediate
`rlwimi[.]`	`rD,rA,UIMM,MB,ME`	Rotate left word immediate and mask insert
`rlwinm[.]`	`rD,rA,UIMM,MB,ME`	Rotate left word immediate and mask
`rlwnm[.]`	`rD,rA,rB,MB,ME`	Rotate left word and mask
`sc`		System call
`slw[.]`	`rD,rA,rB`	Shift left word (logical)
`sraw[.]`	`rD,rA,rB`	Shift right word (arithmetic)
`srawi[.]`	`rD,rA,UIMM`	Shift right immediate (arithmetic)
`srw[.]`	`rD,rA,rB`	Shift right word (logical)
`stb[u,x]`	`rS,d(rA)`	Store byte
`sth[u,x]`	`rS,d(rA)`	Store half-word
`stmw`	`rS,d(rA)`	Store multiple words
`stw[u,x]`	`rS,d(rA)`	Store word
`subf[o,.]`	`rD,rA,rB`	Subtract from
`subfc[o,.]`	`rD,rA,rB`	Subtract from, update carry
`subfe[o,.]`	`rD,rA,rB`	Subtract from, extended
`subfic`	`rD,rA,SIMM`	Subtract from immediate, update carry
`subfme[o,.]`	`rD,rA`	Subtract from -1, extended
`subfze[o,.]`	`rD,rA`	Subtract from 0, extended
`tw`	`TO,rA,rB`	Trap word
`twi`	`TO,rA,SIMM`	Trap word immediate
`xor[.]`	`rD,rA,rB`	XOR
`xorc[.]`	`rD,rA,rB`	XOR with complement
`xori`	`rD,rA,UIMM`	XOR with immediate
`xoris`	`rD,rA,UIMM`	XOR with shifted immediate

NOTE: Many features have been omitted for brevity. This includes Floating-Point operations and registers, supervisor-level operations, string load and stores and ALTIVEC instructions.