Assembler Language Reference
Third Edition (July 2006)

This edition applies to AIX 5L Version 5.3 and to all subsequent releases of this product until otherwise indicated in new editions.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to Information Development, Department 04XA-905-6C006, 11501 Burnet Road, Austin, Texas 78758-3493. To send comments electronically, use this commercial Internet address: aix6kpub@austin.ibm.com. Any information that you supply may be used without incurring any obligation to you.

© Copyright International Business Machines Corporation 1997, 2006. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
# Contents

**About This Book** ........................................................................................................... xi
Highlighting ...................................................................................................................... xi
Case-Sensitivity in AIX ..................................................................................................... xi
ISO 9000 ............................................................................................................................ xi
Related Publications ........................................................................................................ xi

**Chapter 1. Assembler Overview** .................................................................................. 1
Features of the AIX Assembler ....................................................................................... 1
Assembler Installation ..................................................................................................... 9

**Chapter 2. Processing and Storage** ........................................................................... 11
POWER family and PowerPC Architecture Overview ................................................ 11
Branch Processor ............................................................................................................ 19
Fixed-Point Processor .................................................................................................... 21
Floating-Point Processor ................................................................................................. 24

**Chapter 3. Syntax and Semantics** ............................................................................. 27
Character Set .................................................................................................................. 27
Reserved Words .............................................................................................................. 28
Line Format ..................................................................................................................... 28
Statements ...................................................................................................................... 29
Symbols .......................................................................................................................... 31
Constants ......................................................................................................................... 35
Operators ......................................................................................................................... 38
Expressions ..................................................................................................................... 39

**Chapter 4. Addressing** ............................................................................................... 47
Absolute Addressing ....................................................................................................... 47
Absolute Immediate Addressing .................................................................................... 47
Relative Immediate Addressing ...................................................................................... 48
Explicit-Based Addressing .............................................................................................. 48
Implicit-Based Addressing .............................................................................................. 50
Location Counter ............................................................................................................ 51

**Chapter 5. Assembling and Linking a Program** ............................................................ 53
Assembling and Linking a Program ............................................................................. 53
Understanding Assembler Passes .................................................................................. 57
Interpreting an Assembler Listing ................................................................................. 59
Interpreting a Symbol Cross-Reference ........................................................................ 63
Subroutine Linkage Convention .................................................................................... 65
Understanding and Programming the TOC ................................................................... 82
Running a Program ......................................................................................................... 87

**Chapter 6. Extended Instruction Mnemonics** ............................................................. 89
Extended Mnemonics of Branch Instructions ................................................................ 89
Extended Mnemonics of Condition Register Logical Instructions ............................... 96
Extended Mnemonics of Fixed-Point Arithmetic Instructions ....................................... 97
Extended Mnemonics of Fixed-Point Compare Instructions ........................................ 98
Extended Mnemonics of Fixed-Point Load Instructions ................................................. 99
Extended Mnemonics of Fixed-Point Logical Instructions ........................................... 100
Extended Mnemonics of Fixed-Point Trap Instructions ............................................... 100
Extended Mnemonic mtc for Moving to the Condition Register .................................... 102
Extended Mnemonics of Moving from or to Special-Purpose Registers ....................... 102
Chapter 7. Migrating Source Programs

Related Information

Functional Differences for POWER family and PowerPC Instructions

Differences between POWER family and PowerPC Instructions with the Same Op Code

Extended Mnemonics Changes

POWER family Instructions Deleted from PowerPC

Added PowerPC Instructions

Instructions Available Only for the PowerPC 601 RISC Microprocessor

Migration of Branch Conditional Statements with No Separator after Mnemonic

Chapter 8. Instruction Set

abs (Absolute) Instruction

add (Add) or cax (Compute Address) Instruction

adde or a (Add Carrying) Instruction

addi (Add Immediate) or cal (Compute Address Lower) Instruction

addic or ai (Add Immediate Carrying) Instruction

addic or ai. (Add Immediate Carrying and Record) Instruction

addis or cau (Add Immediate Shifted) Instruction

addme or ame (Add to Minus One Extended) Instruction

addze or aze (Add to Zero Extended) Instruction

and (AND) Instruction

andc (AND with Complement) Instruction

andi. or andil. (AND Immediate) Instruction

andis. or andiu. (AND Immediate Shifted) Instruction

cmp (Compare) Instruction

cmpi (Compare Immediate) Instruction

cmpl (Compare Logical) Instruction

cmpli (Compare Logical Immediate) Instruction

cntlz (Count Leading Zeros Word) Instruction

cntlzd (Count Leading Zeros Double Word) Instruction

crand (Condition Register AND) Instruction

crandc (Condition Register AND with Complement) Instruction

crnand (Condition Register NAND) Instruction

cror (Condition Register OR) Instruction

croc (Condition Register OR with Complement) Instruction

cxor (Condition Register XOR) Instruction

dcbf (Data Cache Block Flush) Instruction

dcbi (Data Cache Block Invalidate) Instruction

dcbst (Data Cache Block Store) Instruction

dcbt (Data Cache Block Touch) Instruction

dcbstst (Data Cache Block Touch for Store) Instruction

dcbz or dclz (Data Cache Block Set to Zero) Instruction

dclst (Data Cache Line Store) Instruction

Extended Mnemonics of 32-bit Fixed-Point Rotate and Shift Instructions

Extended Mnemonics of 64-bit Fixed-Point Rotate and Shift Instructions
div (Divide) Instruction .............................................................. 182
divd (Divide Double Word) Instruction ........................................ 184
divdu (Divide Double Word Unsigned) Instruction ........................ 185
divs (Divide Short) Instruction .................................................. 186
divw (Divide Word) Instruction .................................................. 188
divwu (Divide Word Unsigned) Instruction .................................... 190
doz (Difference or Zero) Instruction ............................................ 191
dozi (Difference or Zero Immediate) Instruction ............................. 193
eciwx (External Control In Word Indexed) Instruction ................. 194
ecowx (External Control Out Word Indexed) Instruction .................. 195
eieio (Enforce In-Order Execution of I/O) Instruction ...................... 196
extsw (Extend Sign Word) Instruction ......................................... 197
eqv (Equivalent) Instruction ..................................................... 198
extsb (Extend Sign Byte) Instruction .......................................... 199
extsh or exts (Extend Sign Halfword) Instruction ......................... 200
fabs (Floating Absolute Value) Instruction ................................... 202
fadd or fa (Floating Add) Instruction .......................................... 203
fcfid (Floating Convert from Integer Double Word) Instruction ........ 205
fcmpo (Floating Compare Ordered) Instruction ................................ 206
fcmpu (Floating Compare Unordered) Instruction ............................ 207
fctid (Floating Convert to Integer Double Word) Instruction .......... 208
fctidz (Floating Convert to Integer Double Word with Round toward Zero) Instruction .............................................................. 209
fctiw or fcir (Floating Convert to Integer Word) Instruction .......... 210
fctiwz or fcirz (Floating Convert to Integer Word with Round to Zero) Instruction .............................................................. 212
fdiv or fd (Floating Divide) Instruction ...................................... 213
fmadd or fma (Floating Multiply-Add) Instruction .......................... 216
fmr (Floating Move Register) Instruction ..................................... 218
fmsub or fms (Floating Multiply-Subtract) Instruction ..................... 219
fmul or fm (Floating Multiply) Instruction ................................... 221
fnabs (Floating Negative Absolute Value) Instruction ...................... 224
fneg (Floating Negate) Instruction ............................................. 225
fnmadd or fnma (Floating Negative Multiply-Add) Instruction ........ 226
fnmsub or fnms (Floating Negative Multiply-Subtract) Instruction .... 228
fres (Floating Reciprocal Estimate Single) Instruction .................... 231
frsp (Floating Round to Single Precision) Instruction ..................... 233
frsqrte (Floating Reciprocal Square Root Estimate) Instruction ......... 235
fsel (Floating-Point Select) Instruction ....................................... 236
fsqrt (Floating Square Root, Double-Precision) Instruction .......... 237
fsqrts (Floating Square Root Single) Instruction ........................... 239
fsqrt (Floating Square Root Single) Instruction ........................... 239
fsubs or fs (Floating Subtract) Instruction .................................. 240
icbi (Instruction Cache Block Invalidate) Instruction ................. 242
isync or ics (Instruction Synchronize) Instruction ........................ 243
lbz (Load Byte and Zero) Instruction .......................................... 245
lbzu (Load Byte and Zero with Update) Instruction ....................... 246
lbzux (Load Byte and Zero with Update Indexed) Instruction ........ 247
lbzx (Load Byte and Zero Indexed) Instruction ............................. 248
ld (Load Double Word) Instruction ............................................ 249
ldarx (Store Double Word Reserve Indexed) Instruction ............... 250
ldu (Store Double Word with Update) Instruction ........................ 251
ldux (Store Double Word with Update Indexed) Instruction ............ 252
ldx (Store Double Word Indexed) Instruction .............................. 253
lf (Load Floating-Point Double) Instruction ................................ 253
lfdu (Load Floating-Point Double with Update) Instruction ............. 254
lfdux (Load Floating-Point Double with Update Indexed) Instruction .. 255
lfdx (Load Floating-Point Double-Indexed) Instruction .................... 256
lfq (Load Floating-Point Quad) Instruction .................................. 257
mulli or muli (Multiply Low Immediate) Instruction .................................................. 325
mullw or muls (Multiply Low Word) Instruction .......................................................... 326
nabs (Negative Absolute) Instruction ....................................................................... 328
nand (NAND) Instruction ............................................................................................ 330
neg (Negate) Instruction ............................................................................................... 331
nor (NOR) Instruction .................................................................................................. 333
or (OR) Instruction ....................................................................................................... 334
oric or oril (OR Immediate) Instruction ..................................................................... 336
oris or ori (OR Immediate) Instruction ...................................................................... 337
popcntbd (Population Count Byte Doubleword) Instruction ..................................... 338
rac (Real Address Compute) Instruction ..................................................................... 340
rfi (Return from Interrupt) Instruction ...................................................................... 341
rfid (Return from Interrupt Double Word) Instruction ............................................. 342
rldcl (Rotate Left Double Word then Clear Left) Instruction ...................................... 343
rldcl (Rotate Left Double Word Immediate then Clear Left) Instruction ................. 344
rldcr (Rotate Left Double Word then Clear Right) Instruction ................................... 345
rldic (Rotate Left Double Word Immediate then Clear) Instruction ......................... 346
rldicr (Rotate Left Double Word Immediate then Clear Right) Instruction .............. 347
rldimi (Rotate Left Double Word Immediate then Mask Insert) Instruction .......... 348
rlmi (Rotate Left Then Mask Insert) Instruction ......................................................... 350
rllm or rllm (Rotate Left Word Immediate Then Mask Insert) Instruction ............... 351
rlwinm or rlnm (Rotate Left Word Then AND with Mask) Instruction .................... 352
rlwimi or rlimi (Rotate Left Word Immediate Then Mask Insert) Instruction ........ 353
sc (System Call) Instruction ......................................................................................... 360
si (Subtract Immediate) Instruction ........................................................................... 362
sliq (Shift Left Immediate with MQ) Instruction ....................................................... 367
sllq (Shift Left with MQ) Instruction ......................................................................... 372
slw or sl (Shift Right Word) Instruction ................................................................... 382
sradi (Shift Right Algebraic Double Word) Instruction ............................................. 376
sradi (Shift Right Algebraic Double Word Immediate) Instruction ......................... 377
srq (Shift Right with MQ) Instruction ....................................................................... 386
srq (Shift Right with MQ) Instruction ....................................................................... 387
srw or sr (Shift Right Word) Instruction ................................................................... 394
stb (Store Byte) Instruction ....................................................................................... 396
stbu (Store Byte with Update) Instruction .................................................................. 397
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>stbux (Store Byte with Update Indexed) Instruction</td>
<td>398</td>
</tr>
<tr>
<td>stbx (Store Byte Indexed) Instruction</td>
<td>399</td>
</tr>
<tr>
<td>std (Store Double Word) Instruction</td>
<td>400</td>
</tr>
<tr>
<td>stdcx. (Store Double Word Conditional Indexed)</td>
<td>401</td>
</tr>
<tr>
<td>stdu (Store Double Word with Update) Instruction</td>
<td>402</td>
</tr>
<tr>
<td>stdux (Store Double Word with Update Indexed)</td>
<td>403</td>
</tr>
<tr>
<td>stdx (Store Double Word Indexed) Instruction</td>
<td>404</td>
</tr>
<tr>
<td>stdf (Store Floating-Point Double) Instruction</td>
<td>405</td>
</tr>
<tr>
<td>stdfdu (Store Floating-Point Double with Update)</td>
<td>406</td>
</tr>
<tr>
<td>stdfdux (Store Floating-Point Double with Update Indexed)</td>
<td>407</td>
</tr>
<tr>
<td>stdfx (Store Floating-Point Double Indexed)</td>
<td>408</td>
</tr>
<tr>
<td>stf (Store Floating-Point Quad) Instruction</td>
<td>409</td>
</tr>
<tr>
<td>stfqu (Store Floating-Point Quad with Update)</td>
<td>410</td>
</tr>
<tr>
<td>stfqux (Store Floating-Point Quad with Update Indexed)</td>
<td>411</td>
</tr>
<tr>
<td>stf (Store Floating-Point Quad Indexed) Instruction</td>
<td>412</td>
</tr>
<tr>
<td>stfs (Store Floating-Point Single) Instruction</td>
<td>413</td>
</tr>
<tr>
<td>stfsu (Store Floating-Point Single with Update)</td>
<td>414</td>
</tr>
<tr>
<td>stfsux (Store Floating-Point Single with Update Indexed)</td>
<td>415</td>
</tr>
<tr>
<td>stfsx (Store Floating-Point Single Indexed)</td>
<td>416</td>
</tr>
<tr>
<td>th (Store Half) Instruction</td>
<td>417</td>
</tr>
<tr>
<td>sthbrx (Store Half Byte-Reverse Indexed) Instruction</td>
<td>418</td>
</tr>
<tr>
<td>sthu (Store Half with Update) Instruction</td>
<td>419</td>
</tr>
<tr>
<td>sthux (Store Half with Update Indexed) Instruction</td>
<td>420</td>
</tr>
<tr>
<td>sthx (Store Half Indexed) Instruction</td>
<td>421</td>
</tr>
<tr>
<td>stswi or stsi (Store String Word Immediate)</td>
<td>422</td>
</tr>
<tr>
<td>stswx or stsx (Store String Word Indexed)</td>
<td>423</td>
</tr>
<tr>
<td>stw or st (Store) Instruction</td>
<td>424</td>
</tr>
<tr>
<td>stdx or std (Store Word Indexed)</td>
<td>425</td>
</tr>
<tr>
<td>stwbbrx or sdbbrx (Store Word Byte-Reverse Indexed)</td>
<td>426</td>
</tr>
<tr>
<td>stwcx. (Store Word Conditional Indexed) Instruction</td>
<td>427</td>
</tr>
<tr>
<td>stwu or stu (Store Word with Update) Instruction</td>
<td>428</td>
</tr>
<tr>
<td>stwux or stux (Store Word with Update Indexed)</td>
<td>429</td>
</tr>
<tr>
<td>stwx or stx (Store Word Indexed)</td>
<td>430</td>
</tr>
<tr>
<td>sub (Subtract From) Instruction</td>
<td>431</td>
</tr>
<tr>
<td>subc or sf (Subtract from Carrying) Instruction</td>
<td>432</td>
</tr>
<tr>
<td>subfe or sfe (Subtract from Extended) Instruction</td>
<td>433</td>
</tr>
<tr>
<td>subfic or sfi (Subtract from Immediate Carrying) Instruction</td>
<td>434</td>
</tr>
<tr>
<td>subfme or sfme (Subtract from Minus One Extended) Instruction</td>
<td>435</td>
</tr>
<tr>
<td>sfsz or sfze (Subtract from Zero Extended) Instruction</td>
<td>436</td>
</tr>
<tr>
<td>svc (Supervisor Call) Instruction</td>
<td>437</td>
</tr>
<tr>
<td>sync (Synchronize) or dcs (Data Cache Synchronize) Instruction</td>
<td>438</td>
</tr>
<tr>
<td>td (Trap Double Word) Instruction</td>
<td>439</td>
</tr>
<tr>
<td>tdi (Trap Double Word Immediate) Instruction</td>
<td>440</td>
</tr>
<tr>
<td>tlbiv or tlb (Translation Look-Aside Buffer Invalidate Entry) Instruction</td>
<td>441</td>
</tr>
<tr>
<td>tlbiv (Load Data TLB Entry) Instruction</td>
<td>442</td>
</tr>
<tr>
<td>tlb (Load Instruction TLB Entry) Instruction</td>
<td>443</td>
</tr>
<tr>
<td>tbsync (Translation Look-Aside Buffer Synchronize) Instruction</td>
<td>444</td>
</tr>
<tr>
<td>tw or t (Trap Word) Instruction</td>
<td>445</td>
</tr>
<tr>
<td>twi or ti (Trap Word Immediate) Instruction</td>
<td>446</td>
</tr>
<tr>
<td>xor (XOR) Instruction</td>
<td>447</td>
</tr>
<tr>
<td>xor or xorl (XOR Immediate) Instruction</td>
<td>448</td>
</tr>
<tr>
<td>xoris or xoriu (XOR Immediate Shift) Instruction</td>
<td>449</td>
</tr>
</tbody>
</table>

Chapter 9. Pseudo-ops

viii Assembler Language Reference
Appendix A. Messages .......................................................... 513
Appendix B. Instruction Set Sorted by Mnemonic ..................... 533
Appendix C. Instruction Set Sorted by Primary and Extended Op Code ......................................................... 547
Appendix D. Instructions Common to POWER family, POWER2, and PowerPC ........................................ 561
About This Book

This book is intended for experienced assembler language programmers. Users should be familiar with the AIX® operating system or UNIX® System V commands, assembler instructions, pseudo-ops, and processor register usage. This reference discusses features and specific usage for this version of the Assembler including: installation, operation, syntax, addressing considerations, migration, instructions sets, and pseudo-ops. Also covered are extended mnemonics for POWER-based architectures and their supported processors.

Highlighting

The following highlighting conventions are used in this book:

**Bold** Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.

*Italic* Identifies parameters whose actual names or values are to be supplied by the user.

**Monospace** Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

Case-Sensitivity in AIX

Everything in the AIX 5L operating system is case-sensitive, which means that it distinguishes between uppercase and lowercase letters. For example, you can use the `ls` command to list files. If you type `LS`, the system responds that the command is “not found.” Likewise, `FILEA`, `FiLea`, and `filea` are three distinct file names, even if they reside in the same directory. To avoid causing undesirable actions to be performed, always ensure that you use the correct case.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

Related Publications

The following books contain information about or related to the assembler:

- AIX 5L Version 5.3 Commands Reference Volume 1: a through c
- AIX 5L Version 5.3 Commands Reference Volume 2: d through h
- AIX 5L Version 5.3 Commands Reference Volume 3: i through m
- AIX 5L Version 5.3 Commands Reference Volume 4: n through r
- AIX 5L Version 5.3 Commands Reference Volume 5: s through u
- AIX 5L Version 5.3 Commands Reference Volume 6: v through z
- AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs
Chapter 1. Assembler Overview

The assembler is a program that operates within the operating system. The assembler takes machine-language instructions and translates them into machine object code. The following articles discuss the features of the assembler:

- “Features of the AIX Assembler”
- “Assembler Installation” on page 9

Features of the AIX Assembler

This section describes features of the AIX assembler.

Multiple Hardware Architecture and Implementation Platform Support

The assembler supports the following systems:

- Systems using the first-generation POWER family processors (POWER family architecture)
- Systems using the POWER2 processors (POWER family architecture)
- Systems using the PowerPC 601 RISC Microprocessor, PowerPC 604 RISC Microprocessor, or the PowerPC A35 RISC Microprocessor (PowerPC architecture)
- Systems using POWER4™ processors
- Systems using POWER5™ processors
- Systems using PPC970 processors

The assembler also supports development of programs for the PowerPC 603 RISC Microprocessor (PowerPC architecture).

Attention: The PowerPC 601 RISC Microprocessor implements the PowerPC architecture plus most of the POWER family instructions that are not included in the PowerPC architecture. This implementation provides a POWER family-to-PowerPC bridge processor that runs existing POWER family applications without recompiling and also runs PowerPC applications. Future PowerPC systems might not provide this bridge. An application should not be coded using a mixture of POWER family and PowerPC architecture-unique instructions. Doing so can result in an application that will run only on a PowerPC 601 RISC Microprocessor-based system. Such an application will not run on an existing POWER family machine and is unlikely to run with acceptable performance on future PowerPC machines.

There are several categories of instructions. The following table lists the categories of instructions and shows which implementations support each instruction category. The “X” means the implementation supports the instruction category.

<table>
<thead>
<tr>
<th>Implementations Supporting Each Category of Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Category</td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
<tr>
<td>POWER2-unique instructions</td>
</tr>
<tr>
<td>POWER2 and PowerPC common instructions, not in POWER family</td>
</tr>
<tr>
<td>POWER family-unique instructions not supported by PowerPC 601 RISC Microprocessor</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1997, 2006
<table>
<thead>
<tr>
<th>POWER family-unique instructions supported by PowerPC 601 RISC Microprocessor</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>POWER family and PowerPC common instructions with same mnemonics</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>POWER family and PowerPC common instructions with different mnemonics</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PowerPC instructions supported by PowerPC 601 RISC Microprocessor</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions unique to PowerPC 601 RISC Microprocessor</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC instructions not supported by PowerPC 601 RISC Microprocessor</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 32-bit optional instruction set 1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 32-bit optional instruction set 2</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions unique to PowerPC 603 RISC Microprocessor</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Implementations Supporting Each Category of Instructions

<table>
<thead>
<tr>
<th>Instruction Category</th>
<th>POWER family</th>
<th>POWER2</th>
<th>601</th>
<th>603</th>
<th>604</th>
<th>A35</th>
<th>970</th>
<th>PWR5</th>
</tr>
</thead>
<tbody>
<tr>
<td>POWER2-unique instructions</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POWER2 and PowerPC common instructions, not in POWER family</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>POWER family-unique instructions not supported by PowerPC 601 RISC Microprocessor</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>601</td>
<td>603</td>
<td>604</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------------------------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POWER family-unique instructions supported by PowerPC 601 RISC Microprocessor</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POWER family and PowerPC common instructions with same mnemonics</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>POWER family and PowerPC common instructions with different mnemonics</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PowerPC instructions supported by PowerPC 601 RISC Microprocessor</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions unique to PowerPC 601 RISC Microprocessor</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC instructions not supported by PowerPC 601 RISC Microprocessor</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 32-bit optional instruction set 1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 32-bit optional instruction set 2</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions unique to PowerPC 603 RISC Microprocessor</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 64-bit instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PowerPC Vector instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

The following abbreviations are used in the heading of the previous table:

- **601** PowerPC 601 RISC Microprocessor
- **603** PowerPC 603 RISC Microprocessor
- **604** PowerPC 604 RISC Microprocessor
Host Machine Independence and Target Environment Indicator Flag

The host machine is the hardware platform on which the assembler runs. The target machine is the platform on which the object code is run. The assembler can assemble a source program for any target machine, regardless of the host machine on which the assembler runs.

The target machine can be specified by using either the assembly mode option flag `-m` of the `as` command or the `.machine` pseudo-op. If neither the `-m` flag nor the `.machine` pseudo-op is used, the default assembly mode is used. If both the `-m` flag and a `.machine` pseudo-op are used, the `.machine` pseudo-op overrides the `-m` flag. Multiple `.machine` pseudo-ops are allowed in a source program. The value in a later `.machine` pseudo-op overrides a previous `.machine` pseudo-op.

The default assembly mode provided by the AIX assembler has the POWER family/PowerPC intersection as the target environment, but treats all POWER/PowerPC incompatibility errors (including instructions outside the POWER/PowerPC intersection and invalid form errors) as instructional warnings. The `-W` and `-w` assembler flags control whether these warnings are displayed. In addition to being closed by the absence of the `-m` flag of the `as` command or the `.machine` pseudo-op, the default assembly mode can also be explicitly specified with the `-m` flag of the `as` command or with the `.machine` pseudo-op.

To assemble a source program containing platform-unique instructions from more than one platform without errors or warnings, use one of the following methods:

- Use the `.machine` pseudo-op in the source program.
- Assemble the program with the assembly mode set to the `any` mode (with the `-m` flag of the `as` command).

For example, the source code cannot contain both POWER family-unique instructions and PowerPC 601 RISC Microprocessor-unique instructions. This is also true for each of the sub-source programs contained in a single source program. A sub-source program begins with a `.machine` pseudo-op and ends before the next `.machine` pseudo-op. Since a source program can contain multiple `.machine` pseudo-ops, it normally consists of several sub-source programs. For more information, see the `.machine` pseudo-op.

Mnemonics Cross-Reference

The assembler supports both PowerPC and POWER family mnemonics. The assembler listing has a cross-reference for both mnemonics. The cross-reference is restricted to instructions that have different mnemonics in the POWER family and PowerPC architectures, but which share the same op codes, functions, and operand input formats.

The assembler listing contains a column to display mnemonics cross-reference information. For more information on the assembler listing, see Interpreting an Assembler Listing.

The mnemonics cross-reference helps the user migrate a source program from one architecture to another. The `-s` flag for the `as` command provides a mnemonics cross-reference in the assembler listing to assist with migration. If the `-s` flag is not used, no mnemonics cross-reference is provided.

CPU ID Definition

During the assembly process the assembler determines which instruction set (from a list of several complete instruction sets defined in the architectures or processor implementations) is the smallest instruction set containing all the instructions used in the program. The program is given a CPU ID value indicating this instruction set. Therefore a CPU ID indicates the target environment on which the object code can be run. The CPU ID value for the program is an assembler output value included in the XCOFF object file generated by the assembler.
CPU ID can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com</td>
<td>All instructions used in the program are in the PowerPC and POWER family architecture intersection. (The com instruction set is the smallest instruction set.)</td>
</tr>
<tr>
<td>ppc</td>
<td>All instructions used in the program are in the PowerPC architecture, 32-bit mode, but the program does not satisfy the conditions for CPU ID value com. (The ppc instruction set is a superset of the com instruction set.)</td>
</tr>
<tr>
<td>pwr</td>
<td>All instructions used in the program are in the POWER family architecture, POWER family implementation, but the program does not satisfy the conditions for CPU ID value com. (The pwr instruction set is a superset of the com instruction set.)</td>
</tr>
<tr>
<td>pwr2</td>
<td>All instructions used in the program are in the POWER family architecture, POWER2 implementation, but the program does not satisfy the conditions for CPU ID values com, ppc, or pwr. (The pwr2 instruction set is a superset of the pwr instruction set.)</td>
</tr>
<tr>
<td>any</td>
<td>The program contains a mixture of instructions from the valid architectures or implementations, or contains implementation-unique instructions. The program does not satisfy the conditions for CPU ID values com, ppc, pwr, or pwr2. (The any instruction set is the largest instruction set.)</td>
</tr>
</tbody>
</table>

The assembler output value CPU ID is not the same thing as the assembly mode. The assembly mode (determined by the -m flag of the as command and by use of the .machine pseudo-op in the program) determines which instructions the assembler accepts without errors or warnings. The CPU ID is an output value indicating which instructions are actually used.

In the output XCOFF file, the CPU ID is stored in the low-order byte of the n_type field in a symbol table entry with the C_FILE storage class. The following list shows the low-order byte values and corresponding CPU IDs:

<table>
<thead>
<tr>
<th>Low-Order Byte</th>
<th>CPU ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Not a defined value. An invalid value or object was assembled prior to definition of the CPU-ID field.</td>
</tr>
<tr>
<td>1</td>
<td>ppc</td>
</tr>
<tr>
<td>2</td>
<td>ppc64</td>
</tr>
<tr>
<td>3</td>
<td>com</td>
</tr>
<tr>
<td>4</td>
<td>pwr</td>
</tr>
<tr>
<td>5</td>
<td>any</td>
</tr>
<tr>
<td>18</td>
<td>pwr5</td>
</tr>
<tr>
<td>19</td>
<td>970</td>
</tr>
<tr>
<td>21</td>
<td>vec</td>
</tr>
<tr>
<td>224</td>
<td>pwr2(pwrx)</td>
</tr>
</tbody>
</table>

**Source Language Type**

For cascade compilers, the assembler records the source-language type. In the XCOFF file, the high-order byte of the n_type field of a symbol table entry with the C_FILE storage class holds the source language type information. The following language types are defined:

<table>
<thead>
<tr>
<th>High-Order Byte</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>C</td>
</tr>
<tr>
<td>0x01</td>
<td>FORTRAN</td>
</tr>
<tr>
<td>0x02</td>
<td>Pascal</td>
</tr>
<tr>
<td>0x03</td>
<td>Ada</td>
</tr>
<tr>
<td>0x04</td>
<td>PL/I</td>
</tr>
<tr>
<td>0x05</td>
<td>Basic</td>
</tr>
<tr>
<td>0x06</td>
<td>Lisp</td>
</tr>
<tr>
<td>0x07</td>
<td>Cobol</td>
</tr>
<tr>
<td>0x08</td>
<td>Modula2</td>
</tr>
</tbody>
</table>
The source language type is indicated by the `.source` pseudo-op. By default, the source-language type is “Assembler.” For more information, see the `.source` pseudo-op.

**Detection Error Conditions**

Error number 149 is reported if the source program contains instructions that are not supported in the intended target environment.

An error is reported if the source program contains invalid instruction forms. This error occurs due to incompatibilities between the POWER family and PowerPC architectures. Some restrictions that apply in the PowerPC architecture do not apply in the POWER family architecture. According to the PowerPC architecture, the following invalid instruction forms are defined:

- If an Rc bit, LK bit, or OE bit is defined as `/` (slash) but coded as 1, or is defined as 1 but coded as 0, the form is invalid. Normally, the assembler ensures that these bits contain correct values.

Some fields are defined with more than one `/` (for example, `"//"`). If they are coded as nonzero, the form is invalid. If certain input operands are used for these fields, they must be checked. For this reason, the following instructions are checked:

- For the PowerPC System Call instructions or the POWER family Supervisor Call instructions, if the POWER family `svc` mnemonic is used when the assembly mode is PowerPC type, the SV field must be 0. Otherwise, the instruction form is invalid and error number 165 is reported.

  **Note:** The `svc` and `svcl` instructions are not supported in PowerPC target modes. The `svcla` instruction is supported only on the PowerPC 601 RISC Microprocessor.

- For the Move to Segment Register Indirect instruction, if the POWER family `mtsri` mnemonic is used in PowerPC target modes, the RA field must be 0. Otherwise, the instruction form is invalid and error number 154 is reported. If the PowerPC `mtsrin` mnemonic is used in PowerPC target modes, it requires only two input operands, so no check is needed.

- For all of the Branch Conditional instructions (including Branch Conditional, Branch Conditional to Link Register, and Branch Conditional to Count Register), bits 0-3 of the BO field are checked. If the bits that are required to contain 0 contain a nonzero value, error 150 is reported.

The encoding for the BO field is defined in the section “Branch Processor Instructions” of PowerPC architecture. The following list gives brief descriptions of the possible values for this field:

<table>
<thead>
<tr>
<th>BO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000y</td>
<td>Decrement the Count Register (CTR); then branch if the value of the decremented CTR is not equal to 0 and the condition is False.</td>
</tr>
<tr>
<td>0001y</td>
<td>Decrement the CTR; then branch if the value of the decremented CTR is not equal to 0 and the condition is False.</td>
</tr>
<tr>
<td>001zy</td>
<td>Branch if the condition is False.</td>
</tr>
<tr>
<td>0100y</td>
<td>Decrement the CTR; then branch if the value of the decremented CTR is not equal to 0 and the condition is True.</td>
</tr>
<tr>
<td>0101y</td>
<td>Decrement the CTR; then branch if the value of the decremented CTR is not equal to 0 and the condition is True.</td>
</tr>
<tr>
<td>011zy</td>
<td>Branch if the condition is True.</td>
</tr>
<tr>
<td>1000y</td>
<td>Decrement the CTR; then branch if the value of the decremented CTR is not equal to 0.</td>
</tr>
<tr>
<td>1010y</td>
<td>Decrement the CTR; then branch if the value of the decremented CTR is not equal to 0.</td>
</tr>
<tr>
<td>BO</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>l1z1z</td>
<td>Branch always.</td>
</tr>
</tbody>
</table>

The z bit denotes a bit that must be 0. If the bit is not 0, the instruction form is invalid.

**Note:** The y bit provides a hint about whether a conditional branch is likely to be taken. The value of this bit can be either 0 or 1. The default value is 0. The extended mnemonics for Branch Prediction as defined in PowerPC architecture are used to set this bit to 0 or 1. (See [Extended Mnemonics for Branch Prediction](#) for more information.)

Branch always instructions do not have a y bit in the BO field. Bit 4 of the BO field should contain 0. Otherwise, the instruction form is invalid.

The third bit of the BO field is specified as the "decrement and test CTR" option. For Branch Conditional to Count Register instructions, the third bit of the BO field must not be 0. Otherwise, the instruction form is invalid and error 163 is reported.

- For the update form of fixed-point load instructions, the PowerPC architecture requires that the RA field not be equal to either 0 or the RT field value. Otherwise, the instruction form is invalid and error number 151 is reported.
  
  This restriction applies to the following instructions:
  - `lbzu`
  - `lbzux`
  - `lhzu`
  - `lhsux`
  - `lhau`
  - `lhaux`
  - `lwzu` (*lu* in POWER family)
  - `lwzux` (*lux* in POWER family)

- For the update form of fixed-point store instructions and floating-point load and store instructions, the following instructions require only that the RA field not be equal to 0. Otherwise, the instruction form is invalid and error number 166 is reported.
  
  This restriction applies to the following instructions:
  - `lfsu`
  - `lfsux`
  - `lfdu`
  - `lfdux`
  - `stbu`
  - `stbux`
  - `sthu`
  - `sthux`
  - `stwu` (*stu* in POWER family)
  - `stwux` (*stux* in POWER family)
  - `stfsu`
  - `stfux`
  - `stfdu`
  - `stfdux`

- For multiple register load instructions, the PowerPC architecture requires that the RA field and the RB field, if present in the instruction format, not be in the range of registers to be loaded. Also, RA=RT=0 is not allowed. If RA=RT=0, the instruction form is invalid and error 164 is reported. This restriction applies to the following instructions:
  - `lmn` (*lm* in POWER family)
- lswi (lsi in POWER family)
- lswx (lsx in POWER family)

**Note:** For the lswx instruction, the assembler only checks whether RA=RT=0, because the load register range is determined by the content of the XER register at runtime.

- For fixed-point compare instructions, the PowerPC architecture requires that the L field be equal to 0. Otherwise, the instruction form is invalid and error number 154 is reported. This restriction applies to the following instructions:
  - cmp
  - cmpl
  - cmpli
  - cmpl

**Note:** If the target mode is com, or ppc, the assembler checks the update form of fixed-point load instructions, update form of fixed-point store instructions, update form of floating-point load and store instructions, multiple-register load instructions, and fixed-point compare instructions, and reports any errors. If the target mode is any, pwr, pwr2, or 601, no check is performed.

### Warning Messages

Warning messages are listed when the -w flag is used with the as command. Some warning messages are related to instructions with the same op code for POWER family and PowerPC:

- Several instructions have the same op code in both POWER family and PowerPC architectures, but have different functional definitions. The assembler identifies these instructions and reports warning number 153 when the target mode is com and the -w flag of the as command is used. Because these mnemonics differ functionally, they are not listed in the mnemonics cross-reference of the assembler listing generated when the -s flag is used with the as command. The following table lists these instructions.

<table>
<thead>
<tr>
<th>POWER family</th>
<th>PowerPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcs</td>
<td>sync</td>
</tr>
<tr>
<td>ics</td>
<td>isync</td>
</tr>
<tr>
<td>svca</td>
<td>sc</td>
</tr>
<tr>
<td>mtsri</td>
<td>mtsrin</td>
</tr>
<tr>
<td>lsx</td>
<td>lswx</td>
</tr>
</tbody>
</table>

- The following instructions have the same mnemonics and op code, but have different functional definitions in the POWER family and PowerPC architectures. The assembler cannot check for these, because the differences are not based on the machine the instructions execute on, but rather on what protection domain the instructions are running in.
  - mfsr
  - mfmsr
  - mfdec

### Special-Purpose Register Changes and Special-Purpose Register Field Handling

TID, MQ, SDR0, RTCU, and RTCL are special-purpose registers (SPRs) defined in the POWER family architecture. They are not valid in the PowerPC architecture. However, MQ, RTCU, and RTCL are still available in the PowerPC 601 RISC Microprocessor.
DBATL, DBATU, IBATL, IBATU, TBL, and TBU are SPRs defined in the PowerPC architecture. They are not supported for the PowerPC 601 RISC Microprocessor. The PowerPC 601 RISC Microprocessor uses the BATL and BATU SPRs instead.

The assembler provides the extended mnemonics for “move to or from SPR” instructions. The extended mnemonics include all the SPRs defined in the POWER family and PowerPC architectures. An error is generated if an invalid extended mnemonic is used. The assembler does not support extended mnemonics for any of the following:

- POWER2-unique SPRs (IMR, DABR, DSAR, TSR, and ILCR)
- PowerPC 601 RISC Microprocessor-unique SPRs (HID0, HID1, HID2, HID5, PID, BATL, and BATU)
- PowerPC 603 RISC Microprocessor-unique SPRs (DMISS, DCMP, HASH1, HASH2, IMISS, ICMP, RPA, HID0, and IABR)
- PowerPC 604 RISC Microprocessor-unique SPRs (PIE, HID0, IABR, and DABR)

The assembler does not check the SPR field’s encoding value for the mtsp and mfspr instructions, because the SPR encoding codes could be changed or reused. However, the assembler does check the SPR field’s value range. If the target mode is pwr, pwr2, or com, the SPR field has a 5-bit length and a maximum value of 31. Otherwise, the SPR field has a 10-bit length and a maximum value of 1023.

To maintain source-code compatibility of the POWER family and PowerPC architectures, the assembler assumes that the low-order 5 bits and high-order 5 bits of the SPR number are reversed before they are used as the input operands to the mfspr or mtsp instruction.

**Related Information**

Chapter 1, “Assembler Overview,” on page 1.

“Assembler Installation.”


“Pseudo-ops Overview” on page 463.

The `as` command.


**Assembler Installation**

The AIX assembler is installed with the base operating system, along with commands, files, and libraries for developing software applications.

**Related Information**

The `as` command.

Chapter 2. Processing and Storage

The characteristics of machine architecture and the implementation of processing and storage influence the processor’s assembler language. The assembler supports the various processors that implement the POWER family and PowerPC architectures. The assembler can support both the POWER family and PowerPC architectures because the two architectures share a large number of instructions.

This chapter provides an overview and comparison of the POWER family and PowerPC architectures and tells how data is stored in main memory and in registers. It also discusses the basic functions for both the POWER family and PowerPC instruction sets.

All the instructions discussed in this chapter are nonprivileged. Therefore, all the registers discussed in this chapter are related to nonprivileged instructions. Privileged instructions and their related registers are defined in the PowerPC architecture.

The following processing and storage articles provide an overview of the system microprocessor and tells how data is stored both in main memory and in registers. This information provides some of the conceptual background necessary to understand the function of the system microprocessor’s instruction set and pseudo-ops.

- "POWER family and PowerPC Architecture Overview"
- "Branch Processor" on page 19
- "Fixed-Point Processor" on page 21
- "Floating-Point Processor" on page 24
- Appendix I, “Vector Processor,” on page 597

POWER family and PowerPC Architecture Overview

A POWER family or PowerPC microprocessor contains the sequencing and processing controls for instruction fetch, instruction execution, and interrupt action, and implements the instruction set, storage model, and other facilities defined in the POWER family and PowerPC architectures.

A POWER family or PowerPC microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor. The microprocessor can execute the following classes of instructions:

- Branch instructions
- Fixed-point instructions
- Floating-point instructions

The following diagram illustrates a logical representation of instruction processing for the PowerPC microprocessor.
The following table shows the registers of the PowerPC user instruction set architecture. These registers are in the CPU that are used for 32-bit applications and are available to the user.

<table>
<thead>
<tr>
<th>Register</th>
<th>Bits Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition Register (CR)</td>
<td>0-31</td>
</tr>
<tr>
<td>Link Register (LR)</td>
<td>0-31</td>
</tr>
<tr>
<td>Count Register (CTR)</td>
<td>0-31</td>
</tr>
<tr>
<td>General Purpose Registers 00-31 (GPR)</td>
<td>0-31 for each register</td>
</tr>
<tr>
<td>Fixed-Point Exception Register (XER)</td>
<td>0-31</td>
</tr>
<tr>
<td>Floating-Point Registers 00-31 (FPR)</td>
<td>0-63 for each register</td>
</tr>
<tr>
<td>Floating Point Status and Control Register (FPSCR)</td>
<td>0-31</td>
</tr>
</tbody>
</table>

The following table shows the registers of the POWER family user instruction set architecture. These registers are in the CPU that are used for 32-bit applications and are available to the user.

<table>
<thead>
<tr>
<th>Register</th>
<th>Bits Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition Register (CR)</td>
<td>0-31</td>
</tr>
<tr>
<td>Link Register (LR)</td>
<td>0-31</td>
</tr>
</tbody>
</table>

Figure 1. Logical Processing Model. The process begins at the top with Branch Processing, which branches to either fixed-point or float-point processing. These processes send and receive data from storage. Storage will also send more instructions to Branch Processing at the top of the diagram.
The processing unit is a word-oriented, fixed-point processor functioning in tandem with a doubleword-oriented, floating-point processor. The microprocessor uses 32-bit word-aligned instructions. It provides for byte, halfword, and word operand fetches and stores for fixed point, and word and doubleword operand fetches and stores for floating point. These fetches and stores can occur between main storage and a set of 32 general-purpose registers, and between main storage and a set of 32 floating-point registers.

**Instruction Forms**

All instructions are four bytes long and are word-aligned. Therefore, when the processor fetches instructions (for example, branch instructions), the two low-order bits are ignored. Similarly, when the processor develops an instruction address, the two low-order bits of the address are 0.

Bits 0-5 always specify the op code. Many instructions also have an extended op code (for example, XO-form instructions). The remaining bits of the instruction contain one or more fields. The alternative fields for the various instruction forms are shown in the following:

- **I Form**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>OPCD</td>
</tr>
<tr>
<td>6-29</td>
<td>LI</td>
</tr>
<tr>
<td>30</td>
<td>AA</td>
</tr>
<tr>
<td>31</td>
<td>LK</td>
</tr>
</tbody>
</table>

- **B Form**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>OPCD</td>
</tr>
<tr>
<td>6-10</td>
<td>BO</td>
</tr>
<tr>
<td>11-15</td>
<td>BI</td>
</tr>
<tr>
<td>16-29</td>
<td>BD</td>
</tr>
<tr>
<td>30</td>
<td>AA</td>
</tr>
<tr>
<td>31</td>
<td>LK</td>
</tr>
</tbody>
</table>

- **SC Form**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>OPCD</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>Bits</td>
<td>Value</td>
</tr>
<tr>
<td>--------</td>
<td>-----------</td>
</tr>
<tr>
<td>16-29</td>
<td>///</td>
</tr>
<tr>
<td>30</td>
<td>XO</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

- **D Form**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>OPCD</td>
</tr>
<tr>
<td>6-10</td>
<td>RT, RS, FRT, FRS, TO, or BF, /, and L</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D, SI, or UI</td>
</tr>
</tbody>
</table>

- **DS Form**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>OPCD</td>
</tr>
<tr>
<td>6-10</td>
<td>RT or RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-29</td>
<td>DS</td>
</tr>
<tr>
<td>30-31</td>
<td>XO</td>
</tr>
</tbody>
</table>

- **X Instruction Format**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>OPCD</td>
</tr>
<tr>
<td>6-10</td>
<td>RT, FRT, RS, FRS, TO, BT, or BF, /, and L</td>
</tr>
<tr>
<td>11-15</td>
<td>RA, FRA, SR, SPR, or BFA and //</td>
</tr>
<tr>
<td>16-20</td>
<td>RB, FRB, SH, NB, or U and /</td>
</tr>
<tr>
<td>21-30</td>
<td>XO or EO</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

  - **XL Instruction Format**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>OPCD</td>
</tr>
<tr>
<td>6-10</td>
<td>RT or RS</td>
</tr>
<tr>
<td>11-20</td>
<td>spr or /, FXM and /</td>
</tr>
<tr>
<td>21-30</td>
<td>XO or EO</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

  - **XFX Instruction Format**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>OPCD</td>
</tr>
<tr>
<td>6-10</td>
<td>RT or RS</td>
</tr>
<tr>
<td>11-20</td>
<td>spr or /, FXM and /</td>
</tr>
<tr>
<td>Bits</td>
<td>Value</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
</tr>
<tr>
<td>21-30</td>
<td>XO or EO</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

- **XFL Instruction Format**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>OPCD</td>
</tr>
<tr>
<td>6</td>
<td>/</td>
</tr>
<tr>
<td>7-14</td>
<td>FLM</td>
</tr>
<tr>
<td>15</td>
<td>/</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-30</td>
<td>XO or EO</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

- **XO Instruction Format**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>OPCD</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>XO or EO</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

- **A Form**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>OPCD</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-25</td>
<td>FRC</td>
</tr>
<tr>
<td>26-30</td>
<td>XO</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

- **M Form**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>OPCD</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB or SH</td>
</tr>
<tr>
<td>21-25</td>
<td>MB</td>
</tr>
<tr>
<td>26-30</td>
<td>ME</td>
</tr>
<tr>
<td>Bits</td>
<td>Value</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

For some instructions, an instruction field is reserved or must contain a particular value. This is not indicated in the previous figures, but is shown in the syntax for instructions in which these conditions are required. If a reserved field does not have all bits set to 0, or if a field that must contain a particular value does not contain that value, the instruction form is invalid. See "Detection Error Conditions" on page 6 for more information on invalid instruction forms.

**Split-Field Notation**

In some cases an instruction field occupies more than one contiguous sequence of bits, or occupies a contiguous sequence of bits that are used in permuted order. Such a field is called a *split field*. In the previous figures and in the syntax for individual instructions, the name of a split field is shown in lowercase letters, once for each of the contiguous bit sequences. In the description of an instruction with a split field, and in certain other places where the individual bits of a split field are identified, the name of the field in lowercase letters represents the concatenation of the sequences from left to right. In all other cases, the name of the field is capitalized and represents the concatenation of the sequences in some order, which does not have to be left to right. The order is described for each affected instruction.

**Instruction Fields**

**AA (30)**

Specifies an Absolute Address bit:

0 Indicates an immediate field that specifies an address relative to the current instruction address. For I-form branches, the effective address of the branch target is the sum of the LI field sign-extended to 64 bits (PowerPC) or 32 bits (POWER family) and the address of the branch instruction. For B-form branches, the effective address of the branch target is the sum of the BD field sign-extended to 64 bits (PowerPC) or 32 bits (POWER family) and the address of the branch instruction.

1 Indicates an immediate field that specifies an absolute address. For I-form branches, the effective address of the branch target is the LI field sign-extended to 64 bits (PowerPC) or 32 bits (POWER family). For B-form branches, the effective address of the branch target is the BD field sign-extended to 64 bits (PowerPC) or 32 bits (POWER family).

**BA (11:15)**

Specifies a bit in the Condition Register (CR) to be used as a source.

**BB (16:20)**

Specifies a bit in the CR to be used as a source.

**BD (16:29)**

Specifies a 14-bit signed two’s-complement branch displacement that is concatenated on the right with 0b00 and sign-extended to 64 bits (PowerPC) or 32 bits (POWER family). This is an immediate field.

**BF (6:8)**

Specifies one of the CR fields or one of the Floating-Point Status and Control Register (FPSCR) fields as a target. For POWER family, if i=BF(6:8), then the i field refers to bits i*4 to (i*4)+3 of the register.

**BFA (11:13)**

Specifies one of the CR fields or one of the FPSCR fields as a source. For POWER family, if j=BFA(11:13), then the j field refers to bits j*4 to (j*4)+3 of the register.

**BI (11:15)**

Specifies a bit in the CR to be used as the condition of a branch conditional instruction.
BO (6:10) Specifies options for the branch conditional instructions. The possible encodings for the BO field are:

<table>
<thead>
<tr>
<th>BO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000x</td>
<td>Decrement Count Register (CTR). Branch if the decremented CTR value is not equal to 0 and the condition is false.</td>
</tr>
<tr>
<td>0001x</td>
<td>Decrement CTR. Branch if the decremented CTR value is 0 and the condition is false.</td>
</tr>
<tr>
<td>001xx</td>
<td>Branch if the condition is false.</td>
</tr>
<tr>
<td>0010x</td>
<td>Decrement CTR. Branch if the decremented CTR value is not equal to 0 and the condition is true.</td>
</tr>
<tr>
<td>00101x</td>
<td>Decrement CTR. Branch if the decremented CTR value is equal to 0 and the condition is true.</td>
</tr>
<tr>
<td>011x</td>
<td>Branch if the condition is true.</td>
</tr>
<tr>
<td>1x00x</td>
<td>Decrement CTR. Branch if the decremented CTR value is not equal to 0.</td>
</tr>
<tr>
<td>1x01x</td>
<td>Decrement CTR. Branch if bits 32-63 of the CTR are 0 (PowerPC) or branch if the decremented CTR value is equal to 0 (POWER family).</td>
</tr>
<tr>
<td>1x1xx</td>
<td>Branch always.</td>
</tr>
</tbody>
</table>

BT (6:10) Specifies a bit in the CR or in the FPSCR as the target for the result of an instruction.

D (16:31) Specifies a 16-bit signed two's-complement integer that is sign-extended to 64 bits (PowerPC) or 32 bits (POWER family). This is an immediate field.

EO (21:30) Specifies a 10-bit extended op code used in X-form instructions.

EO’ (22:30) Specifies a 9-bit extended op code used in XO-form instructions.

FL1 (16:19) Specifies a 4-bit field in the svc (Supervisor Call) instruction.

FL2 (27:29) Specifies a 3-bit field in the svc instruction.

FLM (7:14) Specifies a field mask that specifies the FPSCR fields which are to be updated by the mtfsf instruction:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>FPSCR field 0 (bits 00:03)</td>
</tr>
<tr>
<td>8</td>
<td>FPSCR field 1 (bits 04:07)</td>
</tr>
<tr>
<td>9</td>
<td>FPSCR field 2 (bits 08:11)</td>
</tr>
<tr>
<td>10</td>
<td>FPSCR field 3 (bits 12:15)</td>
</tr>
<tr>
<td>11</td>
<td>FPSCR field 4 (bits 16:19)</td>
</tr>
<tr>
<td>12</td>
<td>FPSCR field 5 (bits 20:23)</td>
</tr>
<tr>
<td>13</td>
<td>FPSCR field 6 (bits 24:27)</td>
</tr>
<tr>
<td>14</td>
<td>FPSCR field 7 (bits 28:31)</td>
</tr>
</tbody>
</table>

FRA (11:15) Specifies a floating-point register (FPR) as a source of an operation.

FRB (16:20) Specifies an FPR as a source of an operation.

FRC (21:25) Specifies an FPR as a source of an operation.

FRS (6:10) Specifies an FPR as a source of an operation.

FRT (6:10) Specifies an FPR as the target of an operation.
**FXM (12:19)** Specifies a field mask that specifies the CR fields that are to be updated by the `mtcrf` instruction:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>CR field 0 (bits 00:03)</td>
</tr>
<tr>
<td>13</td>
<td>CR field 1 (bits 04:07)</td>
</tr>
<tr>
<td>14</td>
<td>CR field 2 (bits 08:11)</td>
</tr>
<tr>
<td>15</td>
<td>CR field 3 (bits 12:15)</td>
</tr>
<tr>
<td>16</td>
<td>CR field 4 (bits 16:19)</td>
</tr>
<tr>
<td>17</td>
<td>CR field 5 (bits 20:23)</td>
</tr>
<tr>
<td>18</td>
<td>CR field 6 (bits 24:27)</td>
</tr>
<tr>
<td>19</td>
<td>CR field 7 (bits 28:31)</td>
</tr>
</tbody>
</table>

**I (16:19)** Specifies the data to be placed into a field in the FPSCR. This is an immediate field.

**LEV (20:26)** This is an immediate field in the `svc` instruction that addresses the `svc` routine by `b'1' || LEV || b'00000` if the SA field is equal to 0.

**LI (6:29)** Specifies a 24-bit signed two's-complement integer that is concatenated on the right with `0b00` and sign-extended to 64 bits (PowerPC) or 32 bits (POWER family). This is an immediate field.

**LK (31)** Link bit:

- 0: Do not set the Link Register.
- 1: Set the Link Register. If the instruction is a branch instruction, the address of the instruction following the branch instruction is placed in the Link Register. If the instruction is an `svc` instruction, the address of the instruction following the `svc` instruction is placed into the Link Register.

**MB (21:25) and ME (26:30)** (POWER family) Specifies a 32-bit string. This string consists of a substring of ones surrounded by zeros, or a substring of zeros surrounded by ones. The encoding is:

**MB (21:25)**

Index to start bit of substring of ones.

**ME (26:30)**

Index to stop bit of substring of ones.

Let \( mstart = MB \) and \( mstop = ME \):

- **If** \( mstart < mstop + 1 \) then
  - \( mask(mstart..mstop) = \text{ones} \)
  - \( mask(\text{all other}) = \text{zeros} \)
- **If** \( mstart = mstop + 1 \) then
  - \( mask(0:31) = \text{ones} \)
- **If** \( mstart > mstop + 1 \) then
  - \( mask(mstop+1..mstart-1) = \text{zeros} \)
  - \( mask(\text{all other}) = \text{ones} \)

**NB (16:20)** Specifies the number of bytes to move in an immediate string load or store.

**OPCD (0:5)** Primary op code field.

**OE (21)** Enables setting the OV and SO fields in the XER for extended arithmetic.

**RA (11:15)** Specifies a general-purpose register (GPR) to be used as a source or target.

**RB (16:20)** Specifies a GPR to be used as a source.

**Rc (31)** Record bit:

- 0: Do not set the CR.
- 1: Set the CR to reflect the result of the operation.

For fixed-point instructions, CR bits (0:3) are set to reflect the result as a signed quantity. Whether the result is an unsigned quantity or a bit string can be determined from the EQ bit.

For floating-point instructions, CR bits (4:7) are set to reflect Floating-Point Exception, Floating-Point Enabled Exception, Floating-Point Invalid Operation Exception, and Floating-Point Overflow Exception.
RS (6:10) Specifies a GPR to be used as a source.
RT (6:10) Specifies a GPR to be used as a target.
SA (30) SVC Absolute:
0 \[\text{svc} \text{ routine at address } \text{`1'} \text{ LEV } \text{ b'00000'}\]
1 \[\text{svc} \text{ routine at address } \text{x'1FE0'}\]
SH (16:20) Specifies a shift amount.
SI (16:31) Specifies a 16-bit signed integer. This is an immediate field.
SPR (11:20) Specifies an SPR for the \texttt{mtspr} and \texttt{mfspr} instructions. See the \texttt{mtspr} and \texttt{mfspr} instructions for information on the SPR encodings.
SR (11:15) Specifies one of the 16 Segment Registers. Bit 11 is ignored.
TO (6:10) Specifies the conditions on which to trap. See \texttt{Fixed-Point Trap Instructions} for more information on condition encodings.

<table>
<thead>
<tr>
<th>TO Bit</th>
<th>ANDed with Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Compares less than.</td>
</tr>
<tr>
<td>1</td>
<td>Compares greater than.</td>
</tr>
<tr>
<td>2</td>
<td>Compares equal.</td>
</tr>
<tr>
<td>3</td>
<td>Compares logically less than.</td>
</tr>
<tr>
<td>4</td>
<td>Compares logically greater than.</td>
</tr>
</tbody>
</table>

U (16:19) Used as the data to be placed into the FPSCR. This is an immediate field.
UI (16:31) Specifies a 16-bit unsigned integer. This is an immediate field.
XO (21:30, 22:30, 26:30, or 30) Extended op code field.

**Related Information**

- Chapter 2, “Processing and Storage,” on page 11.
- “Branch Processor.”
- “Fixed-Point Processor” on page 21.
- “Floating-Point Processor” on page 24.

**Branch Processor**

The branch processor has three 32-bit registers that are related to nonprivileged instructions:
- Condition Register
- Link Register
- Count Register

These registers are 32-bit registers. The PowerPC architecture supports both 32- and 64-bit implementations.

For both POWER family and PowerPC, the branch processor instructions include the branch instructions, Condition Register field and logical instructions, and the system call instructions for PowerPC or the supervisor linkage instructions for POWER family.

**Branch Instructions**

Use branch instructions to change the sequence of instruction execution.
Since all branch instructions are on word boundaries, the processor performing the branch ignores bits 30 and 31 of the generated branch target address. All branch instructions can be used in unprivileged state.

A branch instruction computes the target address in one of four ways:

- Target address is the sum of a constant and the address of the branch instruction itself.
- Target address is the absolute address given as an operand to the instruction.
- Target address is the address found in the Link Register.
- Target address is the address found in the Count Register.

Using the first two of these methods, the target address can be computed sufficiently ahead of the branch instructions to prefetch instructions along the target path.

Using the third and fourth methods, prefetching instructions along the branch path is also possible provided the Link Register or the Count Register is loaded sufficiently ahead of the branch instruction.

The branch instructions include Branch Unconditional and Branch Conditional. In the various target forms, branch instructions generally either branch unconditionally only, branch unconditionally and provide a return address, branch conditionally only, or branch conditionally and provide a return address. If a branch instruction has the Link bit set to 1, then the Link Register is altered to store the return address for use by an invoked subroutine. The return address is the address of the instruction immediately following the branch instruction.

The assembler supports various extended mnemonics for branch instructions that incorporate the 80 field only or the 80 field and a partial 81 field into the mnemonics. See "Extended Mnemonics of Branch Instructions" on page 89 for more information.

**System Call Instruction**

The PowerPC system call instructions are called supervisor call instructions in POWER family. Both types of instructions generate an interrupt for the system to perform a service. The system call and supervisor call instructions are:

- "sc (System Call) Instruction" on page 360 (PowerPC)
- "svc (Supervisor Call) Instruction" on page 446 (POWER family)

For more information about how these instructions are different, see "Functional Differences for POWER family and PowerPC Instructions" on page 114.

**Condition Register Instructions**

The condition register instructions copy one CR field to another CR field or perform logical operations on CR bits. The assembler supports several extended mnemonics for the Condition Register instructions. See "Extended Mnemonics of Condition Register Logical Instructions" on page 96 for information on extended mnemonics for condition register instructions.

**Related Information**

- "POWER family and PowerPC Architecture Overview" on page 11.
- "Fixed-Point Processor" on page 21.
- "Floating-Point Processor" on page 24.
- Appendix I, "Vector Processor," on page 597.
Fixed-Point Processor

The PowerPC fixed-point processor uses the following registers for nonprivileged instructions.

- Thirty-two 32-bit General-Purpose Registers (GPRs).
- One 32-bit Fixed-Point Exception Register.

The POWER family fixed-point processor uses the following registers for nonprivileged instructions. These registers are:

- Thirty-two 32-bit GPRs
- One 32-bit Fixed-Point Exception Register
- One 32-bit Multiply-Quotient (MQ) Register

The GPRs are the principal internal storage mechanism in the fixed-point processor.

Fixed-Point Load and Store Instructions

The fixed-point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs. The load instructions compute the EA when moving data. If the storage access does not cause an alignment interrupt or a data storage interrupt, the byte, halfword, or word addressed by the EA is loaded into a target GPR. See "Extended Mnemonics of Fixed-Point Load Instructions" on page 99 for information on extended mnemonics for fixed-point load instructions.

The fixed-point store instructions perform the reverse function. If the storage access does not cause an alignment interrupt or a data storage interrupt, the contents of a source GPR are stored in the byte, halfword, or word in storage addressed by the EA.

In user programs, load and store instructions which access unaligned data locations (for example, an attempt to load a word which is not on a word boundary) will be executed, but may incur a performance penalty. Either the hardware performs the unaligned operation, or an alignment interrupt occurs and an operating system alignment interrupt handler is invoked to perform the unaligned operation.

Fixed-Point Load and Store with Update Instructions

Load and store instructions have an "update" form, in which the base GPR is updated with the EA in addition to the regular move of information from or to memory.

For POWER family load instructions, there are four conditions which result in the EA not being saved in the base GPR:

1. The GPR to be updated is the same as the target GPR. In this case, the updated register contains data loaded from memory.
2. The GPR to be updated is GPR 0.
3. The storage access causes an alignment interrupt.
4. The storage access causes a data storage interrupt.

For POWER family store instructions, conditions 2, 3, and 4 result in the EA not being saved into the base GPR.

For PowerPC load and store instructions, conditions 1 and 2 above result in an invalid instruction form.

In user programs, load and store with update instructions which access an unaligned data location will be performed by either the hardware or the alignment interrupt handler of the underlying operating system. An alignment interrupt will result in the EA not being in the base GPR.
Fixed-Point String Instructions
The Fixed-Point String instructions allow the movement of data from storage to registers or from registers to storage without concern for alignment. These instructions can be used for a short move between arbitrary storage locations or to initiate a long move between unaligned storage fields. Load String Indexed and Store String Indexed instructions of zero length do not alter the target register.

Fixed-Point Address Computation Instructions
There are several address computation instructions in POWER family. These are merged into the arithmetic instructions for PowerPC.

Fixed-Point Arithmetic Instructions
The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers. Several subtract mnemonics are provided as extended mnemonics of addition mnemonics. See “Extended Mnemonics of Fixed-Point Arithmetic Instructions” on page 97 for information on these extended mnemonics.

There are differences between POWER family and PowerPC for all of the fixed-point divide instructions and for some of the fixed-point multiply instructions. To assemble a program that will run on both architectures, the milicode routines for division and multiplication should be used. See “Using Milicode Routines” on page 80 for information on the available milicode routines.

Fixed-Point Compare Instructions
The fixed-point compare instructions algebraically or logically compare the contents of register RA with one of the following:
- The sign-extended value of the SI field
- The UI field
- The contents of register RB

Algebraic comparison compares two signed integers. Logical comparison compares two unsigned integers.

There are different input operand formats for POWER family and PowerPC, for example, the L operand for PowerPC. There are also invalid instruction form restrictions for PowerPC. The assembler checks for invalid instruction forms in PowerPC assembly modes.

Extended mnemonics for fixed-point compare instructions are discussed in “Extended Mnemonics of Fixed-Point Compare Instructions” on page 98.

Fixed-Point Trap Instructions
Fixed-point trap instructions test for a specified set of conditions. Traps can be defined for events that should not occur during program execution, such as an index out of range or the use of an invalid character. If a defined trap condition occurs, the system trap handler is invoked to handle a program interruption. If the defined trap conditions do not occur, normal program execution continues.

The contents of register RA are compared with the sign-extended SI field or with the contents of register RB, depending on the particular trap instruction. In 32-bit implementations, only the contents of the low-order 32 bits of registers RA and RB are used in the comparison.

The comparison results in five conditions that are ANDed with the T0 field. If the result is not 0, the system trap handler is invoked. The five resulting conditions are:

<table>
<thead>
<tr>
<th>TO Field Bit</th>
<th>ANDed with Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Less than</td>
</tr>
<tr>
<td>1</td>
<td>Greater than</td>
</tr>
</tbody>
</table>
Extended mnemonics for the most useful TO field values are provided, and a standard set of codes is provided for the most common combinations of trap conditions. See “Extended Mnemonics of Fixed-Point Trap Instructions” on page 100 for information on these extended mnemonics and codes.

Fixed-Point Logical Instructions

Fixed-point logical instructions perform logical operations in a bit-wise fashion. The extended mnemonics for the no-op instruction and the OR and NOR instructions are discussed in “Extended Mnemonics of Fixed-Point Logical Instructions” on page 100.

Fixed-Point Rotate and Shift Instructions

The fixed-point processor performs rotate operations on data from a GPR. These instructions rotate the contents of a register in one of the following ways:

- The result of the rotation is inserted into the target register under the control of a mask. If the mask bit is 1, the associated bit of the rotated data is placed in the target register. If the mask bit is 0, the associated data bit in the target register is unchanged.
- The result of the rotation is ANDed with the mask before being placed into the target register.

The rotate left instructions allow (in concept) right-rotation of the contents of a register. For 32-bit implementations, an \( n \)-bit right-rotation can be performed by a left-rotation of \( 32 - n \).

The fixed-point shift instructions logically perform left and right shifts. The result of a shift instruction is placed in the target register under the control of a generated mask.

Some POWER family shift instructions involve the MQ register. This register is also updated.

Extended mnemonics are provided for extraction, insertion, rotation, shift, clear, and clear left and shift left operations. See “Extended Mnemonics of 32-bit Fixed-Point Rotate and Shift Instructions” on page 107 for information on these mnemonics.

Fixed-Point Move to or from Special-Purpose Registers Instructions

Several instructions move the contents of one Special-Purpose Register (SPR) into another SPR or into a General-Purpose Register (GPR). These instructions are supported by a set of extended mnemonics that have each SPR encoding incorporated into the extended mnemonic. These include both nonprivileged and privileged instructions.

**Note:** The SPR field length is 10 bits for PowerPC and 5 bits for POWER family. To maintain source-code compatibility for POWER family and PowerPC, the low-order 5 bits and high-order 5 bits of the SPR number must be reversed prior to being used as the input operand to the \texttt{mfspr} instruction or the \texttt{mtspr} instruction. The numbers defined in the encoding tables for the \texttt{mfspr} and \texttt{mtspr} instructions have already had their low-order 5 bits and high-order 5 bits reversed. When using the \texttt{dbx} command to debug a program, remember that the low-order 5 bits and high-order 5 bits of the SPR number are reversed in the output from the \texttt{dbx} command.

There are different sets of SPRs for POWER family and PowerPC. Encodings for the same SPRs are identical for POWER family and PowerPC except for moving from the DEC (Decrement) SPR.
Moving from the DEC SPR is privileged in PowerPC, but nonprivileged in POWER family. One bit in the SPR field is 1 for privileged operations, but 0 for nonprivileged operations. Thus, the encoding number for the DEC SPR for the `mfdec` instruction has different values in PowerPC and POWER family. The DEC encoding number is 22 for PowerPC and 6 for POWER family. If the `mfdec` instruction is used, the assembler determines the DEC encoding based on the current assembly mode. The following list shows the assembler processing of the `mfdec` instruction for each assembly mode value:

- If the assembly mode is `pwr`, `pwr2`, or `601`, the DEC encoding is 6.
- If the assembly mode is `ppc`, `603`, or `604`, the DEC encoding is 22.
- If the default assembly mode, which treats POWER family/PowerPC incompatibility errors as instructional warnings, is used, the DEC encoding is 6. Instructional warning 158 reports that the DEC SPR encoding 6 is used to generate the object code. The warning can be suppressed with the `-W` flag.
- If the assembly mode is `any`, the DEC encoding is 6. If the `-w` flag is used, a warning message (158) reports that the DEC SPR encoding 6 is used to generate the object code.
- If the assembly mode is `com`, an error message reports that the `mfdec` instruction is not supported. No object code is generated. In this situation, the `mfspr` instruction must be used to encode the DEC number.

For more information on SPR encodings, see the "Extended Mnemonics of Moving from or to Special-Purpose Registers" on page 102.

### Related Information

- [Chapter 2, “Processing and Storage,” on page 11.](#)
- [“POWER family and PowerPC Architecture Overview” on page 11.](#)
- [“Branch Processor” on page 19.](#)
- [“Floating-Point Processor.”](#)
- [Appendix I, “Vector Processor,” on page 597](#)

### Floating-Point Processor

The POWER family and PowerPC floating-point processors have the same register set for nonprivileged instructions. The registers are:

- Thirty-two 64-bit floating-point registers
- One 32-bit Floating-Point Status and Control Register (FPSCR)

The floating-point processor provides high-performance execution of floating-point operations. Instructions are provided to perform arithmetic, comparison, and other operations in floating-point registers, and to move floating-point data between storage and the floating-point registers.

PowerPC and POWER2 also support conversion operations in floating-point registers.

### Floating-Point Numbers

A floating-point number consists of a signed exponent and a signed significand, and expresses a quantity that is the product of the signed fraction and the number \(2^{**\text{exponent}}\). Encodings are provided in the data format to represent:

- Finite numeric values
- + or - Infinity
- Values that are "Not a Number" (NaN)
Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. Their encoding permits a variable diagnostic information field. They may be used to indicate uninitialized variables and can be produced by certain invalid operations.

**Interpreting the Contents of a Floating-Point Register**

There are thirty-two 64-bit floating-point registers, numbered from floating-point register 0-31. All floating-point instructions provide a 5-bit field that specifies which floating-point registers to use in the execution of the instruction. Every instruction that interprets the contents of a floating-point register as a floating-point value uses the double-precision floating-point format for this interpretation.

All floating-point instructions other than loads and stores are performed on operands located in floating-point registers and place the results in a floating-point register. The Floating-Point Status and Control Register and the Condition Register maintain status information about the outcome of some floating-point operations.

Load and store double instructions transfer 64 bits of data without conversion between storage and a floating-point register in the floating-point processor. Load single instructions convert a stored single floating-format value to the same value in double floating format and transfer that value into a floating-point register. Store single instructions do the opposite, converting valid single-precision values in a floating-point register into a single floating-format value, prior to storage.

**Floating-Point Load and Store Instructions**

Floating-point load instructions for single and double precision are provided. Double-precision data is loaded directly into a floating-point register. The processor converts single-precision data to double precision prior to loading the data into a floating-point register, since the floating-point registers support only floating-point double-precision operands.

Floating-point store instructions for single and double precision are provided. Single-precision stores convert floating-point register contents to single precision prior to storage.

POWER2 provides load and store floating-point quad instructions. These are primarily to improve the performance of arithmetic operations on large volumes of numbers, such as array operations. Data access is normally a performance bottleneck for these types of operations. These instructions transfer 128 bits of data, rather than 64 bits, in one load or store operation (that is, one storage reference). The 128 bits of data is treated as two doubleword operands, not as one quadword operand.

**Floating-Point Move Instructions**

Floating-point move instructions copy data from one FPR to another, with data modification as described for each particular instruction. These instructions do not modify the FPSCR.

**Floating-Point Arithmetic Instructions**

Floating-point arithmetic instructions perform arithmetic operations on floating-point data contained in floating-point registers.

**Floating-Point Multiply-Add Instructions**

Floating-point multiply-add instructions combine a multiply operation and an add operation without an intermediate rounding operation. The fractional part of the intermediate product is 106 bits wide, and all 106 bits are used in the add or subtract portion of the instruction.
Floating-Point Compare Instructions

Floating-point compare instructions perform ordered and unordered comparisons of the contents of two FPRs. The CR field specified by the 8F field is set based on the result of the comparison. The comparison sets one bit of the designated CR field to 1, and sets all other bits to 0. The Floating-Point Condition Code (FPCC) (bits 16:19) is set in the same manner.

The CR field and the FPCC are interpreted as follows:

| Condition-Register Field and Floating-Point Condition Code Interpretation |
|-----------------|------------------|
| Bit | Name | Description |
| 0 | FL | (FRA) < (FRB) |
| 1 | FG | (FRA) > (FRB) |
| 2 | FE | (FRA) = (FRB) |
| 3 | FU | (FRA) ? (FRB) (unordered) |

Floating-Point Conversion Instructions

Floating-point conversion instructions are only provided for PowerPC and POWER2. These instructions convert a floating-point operand in an FPR into a 32-bit signed fixed-point integer. The CR1 field and the FPSCR are altered.

Floating-Point Status and Control Register Instructions

Floating-Point Status and Control Register Instructions manipulate data in the FPSCR.

Related Information

Chapter 2, “Processing and Storage,” on page 11.
"POWER family and PowerPC Architecture Overview” on page 11.
"Branch Processor” on page 19.
"Fixed-Point Processor” on page 21.
Appendix I, “Vector Processor,” on page 597
Chapter 3. Syntax and Semantics

This overview explains the syntax and semantics of assembler language, including the following items:

- "Character Set"
- "Reserved Words" on page 28
- "Line Format" on page 28
- "Statements" on page 29
- "Symbols" on page 31
- "Constants" on page 35
- "Operators" on page 38
- "Expressions" on page 39

Character Set

All letters and numbers are allowed. The assembler discriminates between uppercase and lowercase letters. To the assembler, the variables Name and name identify distinct symbols.

Some blank spaces are required, while others are optional. The assembler allows you to substitute tabs for spaces.

The following characters have special meaning in the operating system assembler language:

- **, (comma)**  
  Operand separator. Commas are allowed in statements only between operands, for example:
  
  a 3,4,5

- **# (pound sign)**  
  Comments. All text following a # to the end of the line is ignored by the assembler. A # can be the first character in a line, or it can be preceded by any number of characters, blank spaces, or both. For example:
  
  a 3,4,5 # Puts the sum of GPR4 and GPR5 into GPR3.

- **: (colon)**  
  Defines a label. The : always appears immediately after the last character of the label name and defines a label equal to the value contained in the location counter at the time the assembler encounters the label. For example:
  
  add: a 3,4,5  # Puts add equal to the address  
  # where the a instruction is found.

- **; (semicolon)**  
  Instruction separator. A semicolon separates two instructions that appear on the same line. Spaces around the semicolon are optional. A single instruction on one line does not have to end with a semicolon.

  To keep the assembler listing clear and easily understandable, it is suggested that each line contain only one instruction. For example:
  
  a 3,4,5  # These two lines have  
  a 4,3,5  # the same effect as...

  a 3,4,5; a 4,3,5  # ...this line.

- **$ (dollar sign)**  
  Refers to the current value in the assembler's current location counter. For example:
  
  dino: .long 1,2,3  
  size: .long $ - dino

Related Information

- "Reserved Words" on page 28
- "Line Format" on page 28
Reserved Words

There are no reserved words in the operating system assembler language. The mnemonics for instructions and pseudo-ops are not reserved. They can be used in the same way as any other symbols.

There may be restrictions on the names of symbols that are passed to programs written in other languages.

Related Information

"Character Set" on page 27
"Line Format"
"Statements" on page 29
"Symbols" on page 31
"Constants" on page 35
"Operators" on page 38
"Expressions" on page 39

Line Format

The assembler supports a free-line format for source lines, which does not require that items be in a particular column position.

For all instructions, a separator character (space or tab) is recommended between the mnemonic and operands of the statement for readability. With the AIX assembler, Branch Conditional instructions need a separator character (space or tab) between the mnemonic and operands for unambiguous processing by the assembler. (See "Migration of Branch Conditional Statements with No Separator after Mnemonic" on page 121 for more information.)
The assembler language puts no limit on the number of characters that can appear on a single input line. If a code line is longer than one line on a terminal, line wrapping will depend on the editor used. However, the listing will only display 512 ASCII characters per line.

Blank lines are allowed; the assembler ignores them.

**Related Information**
- "Character Set" on page 27
- "Reserved Words" on page 28
- "Statements" on page 31
- "Symbols" on page 35
- "Constants" on page 35
- "Operators" on page 38
- "Expressions" on page 39

The `atof` subroutine.


**Statements**

The assembler language has three kinds of statements: instruction statements, pseudo-operation statements, and null statements. The assembler also uses separator characters, labels, mnemonics, operands, and comments.

**Instruction Statements and Pseudo-Operation Statements**

An instruction or pseudo-op statement has the following syntax:

```
[label:] mnemonic [operand1[,operand2...]] [ # comment]
```

The assembler recognizes the end of a statement when one of the following appears:
- An ASCII new-line character
- A # (pound sign) (comment character)
- A ; (semicolon)

**Null Statements**

A null statement does not have a mnemonic or any operands. It can contain a label, a comment, or both. Processing a null statement does not change the value of the location counter.

Null statements are useful primarily for making assembler source code easier for people to read.

A null statement has the following syntax:

```
[label:] [ # comment]
```

The spaces between the label and the comment are optional.
If the null statement has a label, the label receives the value of the next statement, even if the next statement is on a different line. The assembler gives the label the value contained in the current location counter. For example:

```
here:
   a 3,4,5
```

is synonymous with

```
here:  a 3,4,5
```

**Note:** Certain pseudo-ops (\texttt{.csect}, \texttt{.comm}, and \texttt{.lcomm}, for example) may prevent a null statement's label from receiving the value of the address of the next statement.

**Separator Characters**

The separator characters are spaces, tabs, and commas. Commas separate operands. Spaces or tabs separate the other parts of a statement. A tab can be used wherever a space is shown in this book.

The spaces shown in the syntax of an instruction or pseudo-op are required.

Branch Conditional instructions need a separator character (space or tab) between the mnemonic and operands for unambiguous processing by the assembler. (See \texttt{Migration of Branch Conditional Statements with No Separator after Mnemonic} on page 121 for more information.)

Optionally, you can put one or more spaces after a comma, before a pound sign (#), and after a #.

**Labels**

The label entry is optional. A line may have zero, one, or more labels. Moreover, a line may have a label but no other contents.

To define a label, place a symbol before the : (colon). The assembler gives the label the value contained in the assembler's current location counter. This value represents a relocatable address. For example:

```
subtr:  sf 3,4,5
# The label subtr: receives the value
# of the address of the sf instruction.
# You can now use subtr in subsequent statements
# to refer to this address.
```

If the label is in a statement with an instruction that causes data alignment, the label receives its value before the alignment occurs. For example:

```
# Assume that the location counter now
# contains the value of 98.
place:  .long expr
# When the assembler processes this statement, it
# sets place to address 98. But the .long is a pseudo-op that
# aligns expr on a fullword. Thus, the assembler puts
# expr at the next available fullword boundary, which is
# address 100. In this case, place is not actually the address
# at which expr is stored; referring to place will not put you
# at the location of expr.
```

**Mnemonics**

The mnemonic field identifies whether a statement is an instruction statement or a pseudo-op statement. Each mnemonic requires a certain number of operands in a certain format.

For an instruction statement, the mnemonic field contains an abbreviation like \texttt{ai} (Add Immediate) or \texttt{sf} (Subtract From). This mnemonic describes an operation where the system microprocessor processes a
single machine instruction that is associated with a numerical operation code (op code). All instructions are 4 bytes long. When the assembler encounters an instruction, the assembler increments the location counter by the required number of bytes.

For a pseudo-op statement, the mnemonic represents an instruction to the assembler program itself. There is no associated op code, and the mnemonic does not describe an operation to the processor. Some pseudo-ops increment the location counter; others do not. See the “Pseudo-ops Overview” on page 463 for a list of pseudo-ops that change the location counter.

Operands
The existence and meaning of the operands depends on the mnemonic used. Some mnemonics do not require any operands. Other mnemonics require one or more operands.

The assembler interprets each operand in context with the operand’s mnemonic. Many operands are expressions that refer to registers or symbols. For instruction statements, operands can be immediate data directly assembled into the instruction.

Comments
Comments are optional and are ignored by the assembler. Every line of a comment must be preceded by a # (pound sign); there is no other way to designate comments.

Related Information
“Character Set” on page 27
“Reserved Words” on page 28
“Line Format” on page 28
“Symbols”
“Constants” on page 35
“Operators” on page 38
“Expressions” on page 39
The `atof` subroutine.

Symbols
A symbol is a single character or combination of characters used as a label or operand.

Constructing Symbols
Symbols may consist of numeric digits, underscores, periods, uppercase or lowercase letters, or any combination of these. The symbol cannot contain any blanks or special characters, and cannot begin with a digit. Uppercase and lowercase letters are distinct.

If a symbol must contain blank or special characters because of external references, the `rename` pseudo-op can be used to treat a local name as a synonym or alias for the external reference name.
From the assembler's and loader's perspective, the length of a symbol name is limited only by the amount of storage you have.

**Note:** Other routines linked to the assembler language files may have their own constraints on symbol length.

With the exception of control section (csect) or Table of Contents (TOC) entry names, symbols may be used to represent storage locations or arbitrary data. The value of a symbol is always a 32-bit quantity.

The following are valid examples of symbol names:

- READER
- XC2345
- result.a
- resultA
- balance_old
- _label9
- .myspot

The following are not valid symbol names:

- 7_sum  (Begins with a digit.)
- #ofcredits  (The # makes this a comment.)
- aa*1  (Contains *, a special character.)
- IN AREA  (Contains a blank.)

You can define a symbol by using it in one of two ways:

- As a label for an instruction or pseudo-op
- As the name operand of a `.set`, `.comm`, `.lcomm`, `.dsect`, `.csect`, or `.rename` pseudo-op

**Defining a Symbol with a Label**

You can define a symbol by using it as a label. For example:

```assembly
.loop:       .using      dataval[RW],5
             bgt         cont
             .
             bdz         loop
.cont:       1          3,dataval
             a          4,3,4
             .
.glb dataval[RW]
.dataval:    .short     10
```

The assembler gives the value of the location counter at the instruction or pseudo-op's leftmost byte. In the example here, the object code for the `l` instruction contains the location counter value for `dataval`.

At run time, an address is calculated from the `dataval` label, the offset, and GPR 5, which needs to contain the address of `csect dataval[RW]`. In the example, the `l` instruction uses the 16 bits of data stored at the `dataval` label's address.

The value referred to by the symbol actually occupies a memory location. A symbol defined by a label is a relocatable value.
The symbol itself does not exist at run time. However, you can change the value at the address represented by a symbol at run time if some code changes the contents of the location represented by the dataval label.

**Defining a Symbol with a Pseudo-op**

Use a symbol as the name operand of a `.set` pseudo-op to define the symbol. This pseudo-op has the format:

```
.set name,exp
```

The assembler evaluates the `exp` operand, then assigns the value and type of the `exp` operand to the symbol `name`. When the assembler encounters that symbol in an instruction, the assembler puts the symbol's value into the instruction’s object code.

For example:

```
.set number,10
.ai 4,4,number
```

In the preceding example, the object code for the `ai` instruction contains the value assigned to `number`, that is, 10.

The value of the symbol is assembled directly into the instruction and does not occupy any storage space. A symbol defined with a `.set` pseudo-op can have an absolute or relocatable type, depending on the type of the `exp` operand. Also, because the symbol occupies no storage, you cannot change the value of the symbol at run time; reassembling the file will give the symbol a new value.

A symbol also can be defined by using it as the name operand of a `.comm`, `.lcomm`, `.csect`, `.dsect`, or `.rename` pseudo-op. Except in the case of the `.dsect` pseudo-op, the value assigned to the symbol describes storage space.

**CSECT Entry Names**

A symbol can also be defined when used as the `qualname` operand of the `.csect` pseudo-op. When used in this context, the symbol is defined as the name of a csect with the specified storage mapping class. Once defined, the symbol takes on a storage mapping class that corresponds to the name qualifier.

A `qualname` operand takes the form of:

```
symbol[XX]
```

OR

```
symbol{XX}
```

where `XX` is the storage mapping class.

For more information, see the "`.csect Pseudo-op" on page 473."

**The Special Symbol TOC**

Provisions have been made for the special symbol TOC. In XCOFF format modules, this symbol is reserved for the TOC anchor, or the first entry in the TOC. The symbol TOC has been predefined in the assembler so that the symbol TOC can be referred to if its use is required. The `.toc` pseudo-op creates the TOC anchor entry. For example, the following data declaration declares a word that contains the address of the beginning of the TOC:

```
...
.long TOC[TC0]

This symbol is undefined unless a .toc pseudo-op is contained within the assembler file.

For more information, see the ".toc Pseudo-op" on page 504.

**TOC Entry Names**

A symbol can be defined when used as the *Name* operand of the .tc pseudo-op. When used in this manner, the symbol is defined as the name of a TOC entry with a storage mapping class of TC.

The *Name* operand takes the form of:

`symbol[TC]`

For more information, see the ".tc Pseudo-op" on page 503.

**Using a Symbol before Defining It**

It is possible to use a symbol before you define it. Using a symbol and then defining it later in the same file is called *forward referencing*. For example, the following is acceptable:

```
# Assume that GPR 6 contains the address of .csect data[RW].
   15,ten(s)
   ...
   .csect data[RW]
   ten: .long 10
```

If the symbol is not defined in the file in which it occurs, it may be an external symbol or an undefined symbol. When the assembler finds undefined symbols, it gives an error message unless the `-u` flag of the `as` command is used to suppress this error message. External symbols may be declared in a statement using the ".*extern Pseudo-op" on page 481.

**Declaring an External Symbol**

If a local symbol is used that is defined in another module, the .extern pseudo-op is used to declare that symbol in the local file as an external symbol. Any undefined symbols that do not appear in a statement with the .extern or .globl pseudo-op will be flagged with an error.

**Related Information**

- "Character Set" on page 27
- "Reserved Words" on page 28
- "Line Format" on page 28
- "Statements" on page 29
- "Constants" on page 35
- "Operators" on page 38
- "Expressions" on page 39

The `atof` subroutine.
Constants
The assembler language provides four kinds of constants:

- Arithmetic constants
- "Character Constants" on page 37
- "Symbolic Constants" on page 37
- "String Constants" on page 37

When the assembler encounters an arithmetic or character constant being used as an instruction's operand, the value of that constant is assembled into the instruction. When the assembler encounters a symbol being used as a constant, the value of the symbol is assembled into the instruction.

Arithmetic Constants
The assembler language provides four kinds of arithmetic constants:

- Decimal
- Octal
- Hexadecimal
- Floating point

In 32-bit mode, the largest signed positive integer number that can be represented is the decimal value \((2^{31}) - 1\). The largest negative value is \(-(2^{31})\). In 64-bit mode, the largest signed positive integer number that can be represented is \((2^{63})-1\). The largest negative value is \(-2^{63}\). Regardless of the base (for example, decimal, hexadecimal, or octal), the assembler regards integers as 32-bit constants.

The interpretation of a constant is dependent upon the assembly mode. In 32-bit mode, the AIX assembler behaves in the same manner as earlier AIX versions: the assembler regards integers as 32-bit constants. In 64-bit mode, all constants are interpreted as 64-bit values. This may lead to results that differ from expectations. For example, in 32-bit mode, the hexadecimal value \(0xFFFFFFFF\) is equivalent to the decimal value of \(-1\). In 64-bit mode, however, the decimal equivalent is 4294967295. To obtain the value \(-1\) the hexadecimal constant \(0xFFFF_FFFF_FFFF_FFFF\) (or the octal equivalent), or the decimal value \(-1\), should be used.

In both 32-bit and 64-bit mode, the result of integer expressions may be truncated if the size of the target storage area is too small to contain an expression result. (In this context, truncation refers to the removal of the excess most-significant bits.)

To improve readability of large constants, especially 64-bit values, the assembler will accept constants containing the underscore ("_") character. The underscore may appear anywhere within the number except the first numeric position. For example, consider the following table:

<table>
<thead>
<tr>
<th>Constant Value</th>
<th>Valid/Invalid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1_800_500</td>
<td>Valid</td>
</tr>
<tr>
<td>0xFFFFFFFFF_00000000</td>
<td>Valid</td>
</tr>
<tr>
<td>0b111010_00100_00101_000000000001000_00</td>
<td>Valid (this is the &quot;ld 4,8(5)&quot; instruction)</td>
</tr>
<tr>
<td>0x_FFFF</td>
<td>Invalid</td>
</tr>
</tbody>
</table>
The third example shows a binary representation of an instruction where the underscore characters are used to delineate the various fields within the instruction. The last example contains a hexadecimal prefix, but the character immediately following is not a valid digit; the constant is therefore invalid.

**Arithmetic Evaluation**

In 32-bit mode, arithmetic evaluation takes place using 32-bit math. For the `.llong` pseudo-op, which is used to specify a 64-bit quantity, any evaluation required to initialize the value of the storage area uses 32-bit arithmetic.

For 64-bit mode, arithmetic evaluation uses 64-bit math. No sign extension occurs, even if a number might be considered negative in a 32-bit context. Negative numbers must be specified using decimal format, or (for example, in hexadecimal format) by using a full complement of hexadecimal digits (16 of them).

**Decimal Constants**

Base 10 is the default base for arithmetic constants. If you want to specify a decimal number, type the number in the appropriate place:

```plaintext
ai 5,4,10  # Add the decimal value 10 to the contents  
# of GPR 4 and put the result in GPR 5.
```

Do not prefix decimal numbers with a 0. A leading zero indicates that the number is octal.

**Octal Constants**

To specify that a number is octal, prefix the number with a `0`:

```plaintext
ai 5,4,0377  # Add the octal value 0377 to the contents  
# of GPR 4 and put the result in GPR 5.
```

**Hexadecimal Constants**

To specify a hexadecimal number, prefix the number with `0X` or `0x`. You can use either uppercase or lowercase for the hexadecimal numerals A through F.

```plaintext
ai 5,4,0xF  # Add the hexadecimal value 0xF to the contents of GPR 4 and put the result in GPR 5.
```

**Binary Constants**

To specify a binary number, prefix the number with `0B` or `0b`.

```plaintext
ori 3,6,0b0010_0001  # OR (the decimal value) 33 with the contents of GPR 6 and put the result in GPR 3.
```

**Floating-Point Constants**

A floating-point constant has the following components in the specified order:

- **Integer Part**: Must be one or more digits.
- **Decimal Point**: (period). Optional if no fractional part follows.
- **Fraction Part**: Must be one or more digits. The fraction part is optional.
- **Exponent Part**: Optional. Consists of an e or E, possibly followed by a + or -, followed by one or more digits.

For assembler input, you can omit the fraction part. For example, the following are valid floating-point constants:

- `0.45`
- `1e+5`
• 4E-11
• 0.99E6
• 357.22e12

Floating-point constants are allowed only where fcon expressions are found.

There is no bounds checking for the operand.

Note: In AIX 4.3 and later, the assembler uses the strtdlo subroutine to perform the conversion to floating point. Check current documentation for restrictions and return values.

Character Constants
To specify an ASCII character constant, prefix the constant with a ’ (single quotation mark). Character constants can appear anywhere an arithmetic constant is allowed, but you can only specify one character constant at a time. For example ‘A’ represents the ASCII code for the character A.

Character constants are convenient when you want to use the code for a particular character as a constant, for example:

cal 3,’X(0)
# Loads GPR 3 with the ASCII code for
# the character X (that is, 0x58).
# After the cal instruction executes, GPR 3 will
# contain binary
# 0x0000 0000 0000 0000 0000 0000 0101 1000.

Symbolic Constants
A symbol can be used as a constant by giving the symbol a value. The value can then be referred to by the symbol name, instead of by using the value itself.

Using a symbol as a constant is convenient if a value occurs frequently in a program. Define the symbolic constant once by giving the value a name. To change its value, simply change the definition (not every reference to it) in the program. The changed file must be reassembled before the new symbol constant is valid.

A symbolic constant can be defined by using it as a label or by using it in a .set statement.

String Constants
String constants differ from other types of constants in that they can be used only as operands to certain pseudo-ops, such as the rename, byte or string pseudo-ops.

The syntax of string constants consists of any number of characters enclosed in “” (double quotation marks):

"any number of characters"

To use a " in a string constant, use double quotation marks twice. For example:

"a double quote character is specified like this ""

Related Information

"Character Set" on page 27
"Reserved Words" on page 28
"Line Format" on page 28
Operators

All operators evaluate from left to right except for the unary operators, which evaluate from right to left.

The assembler provides the following unary operators:

+ unary positive
- unary negative
~ one’s complement (unary)

The assembler provides the following binary operators:

* multiplication
/ division
> right shift
< left shift
I bitwise inclusive or
& bitwise AND
^ bitwise exclusive or
+ addition
- subtraction

Parentheses can be used in expressions to change the order in which the assembler evaluates the expression. Operations within parentheses are performed before operations outside parentheses. Where nested parentheses are involved, processing starts with the innermost set of parentheses and proceeds outward.

Operator Precedence

Operator precedence for 32-bit expressions is shown in the following figure.

Highest Priority

( unary -, unary +, ~
  * / < >
  ^ &

Lowest Priority
In 32-bit mode, all the operators perform 32-bit signed integer operations. In 64-bit mode, all computations are performed using 64-bit signed integer operations.

The division operator produces an integer result; the remainder has the same sign as the dividend. For example:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>8/3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8/-3</td>
<td>-2</td>
<td>2</td>
</tr>
<tr>
<td>(-8)/3</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>(-8)/(-3)</td>
<td>2</td>
<td>-2</td>
</tr>
</tbody>
</table>

The left shift (<) and right shift (>) operators take an integer bit value for the right-hand operand. For example:

```
.set mydata,1
.set newdata,mydata<2
# Shifts 1 left 2 bits.
# Assigns the result to newdata.
```

Related Information

- "Character Set" on page 27
- "Reserved Words" on page 28
- "Line Format" on page 28
- "Statements" on page 29
- "Symbols" on page 31
- "Constants" on page 35
- "Expressions" on page 35

The `atof` subroutine.

```
```

Expressions

A term is the smallest element that the assembler parser can recognize when processing an expression. Each term has a value and a type. An expression is formed by one or more terms. The assembler evaluates each expression into a single value, and uses that value as an operand. Each expression also has a type. If an expression is formed by one term, the expression has the same type as the type of the term. If an expression consists of more than one term, the type is determined by the expression handler according to certain rules applied to all the types of terms contained in the expression. Expression types are important because:

- Some pseudo-ops and instructions require expressions with a particular type.
- Only certain operators are allowed in certain types of expressions.
Object Mode Considerations

One aspect of assembly language expressions is that of the object mode and relocation vs. the size of the data value being calculated. In 32-bit mode, relocation is applied to 32-bit quantities; expressions resulting in a requirement for relocation (for example, a reference to an external symbol) can not have their value stored in any storage area other than a word. For the .long pseudo-op, it is worthwhile to point out that expressions used to initialize the contents of a .long may not require relocation. In 64-bit mode, relocation is applied to double-word quantities. Thus, expression results that require relocation can not have their value stored in a location smaller than a double-word.

Arithmetic evaluations of expressions in 32-bit mode is consistent with the behavior found in prior releases of the assembler. Integer constants are considered to be 32-bit quantities, and the calculations are 32-bit calculations. In 64-bit mode constants are 64-bit values, and expressions are evaluated using 64-bit calculations.

Types and Values of Terms

The following is a list of all the types of terms and an abbreviated name for each type:

- Absolute (E_ABS)
- Relocatable (E_REL)
- External relocatable (E_EXT)
- TOC-relative relocatable (E_TREL)
- TOCOF relocatable (E_TOCOF)

Absolute Terms

A term is absolute if its value does not change upon program relocation. In other words, a term is absolute if its value is independent of any possible code relocation operation.

An absolute term is one of the following items:

- A constant (including all the kinds of constants defined in "Constants" on page 35).
- A symbol set to an absolute expression.

The value of an absolute term is the constant value.

Relocatable Terms

A term is relocatable if its value changes upon program relocation. The value of a relocatable term depends on the location of the control section containing it. If the control section moves to a different storage location (for example, a csect is relocated by the binder at bind time), the value of the relocatable term changes accordingly.

A relocatable term is one of the following items:

- A label defined within a csect that does not have TD or TC as its Storage Mapping Class (SMC)
- A symbol set to a relocatable expression
- A label defined within a dsect
- A dsect name
- A location counter reference (which uses $, the dollar sign)

If it is not used as a displacement for a D-form instruction, the value of a csect label or a location counter reference is its relocatable address, which is the sum of the containing csect address and the offset relative to the containing csect. If it is used as a displacement for a D-form instruction, the assembler implicitly subtracts the containing csect address so that only the offset is used for the displacement. A csect address is the offset relative to the beginning of the first csect of the file.
A dsect is a reference control section that allows you to describe the layout of data in a storage area without actually reserving any storage. A dsect provides a symbolic format that is empty of data. The assembler does assign location counter values to the labels that are defined in a dsect. The values are the offsets relative to the beginning of the dsect. The data in a dsect at run time can be referenced symbolically by using the labels defined in a dsect.

Relocatable terms based on a dsect location counter (either the dsect name or dsect labels) are meaningful only in the context of a .using statement. Since this is the only way to associate a base address with a dsect, the addressability of the dsect is established in combination with the storage area.

A relocatable term may be based on any control section, either csect or dsect, in all the contexts except if it is used as a relocatable address constant. If a csect label is used as an address constant, it represents a relocatable address, and its value is the offset relative to the csect plus the address of the csect. A dsect label cannot be used as a relocatable address constant since a dsect is only a data template and has no address.

If two dsect labels are defined in the same dsect, their difference can be used as an absolute address constant.

**External Relocatable Terms**

A term is external relocatable (E_EXT) if it is an external symbol (a symbol not defined, but declared within the current module, or defined in the current module and globally visible), a csect name, or a TOC entry name.

This term is relocatable because its value will change if it, or its containing control section, is relocated.

An external relocatable term or expression cannot be used as the operand of a .set pseudo-op.

An external relocatable term is one of the following items:
- A symbol defined with the .comm pseudo-op
- A symbol defined with the .lcomm pseudo-op
- A csect name
- A symbol declared with the .globl pseudo-op
- A TOC entry name
- An undefined symbol declared with the .extern pseudo-op

Except for the undefined symbol, if this term is not used as a displacement for a D-form instruction, its value is its relocatable address, which is the offset relative to the beginning of the first csect in the file. If it is used as a displacement for a D-form instruction, the assembler implicitly subtracts the containing csect address (except for a TOC entry name), usually producing a zero displacement because the csect address is subtracted from itself. If a TOC entry name is used as a displacement for a D-form instruction, the assembler implicitly subtracts the address of the TOC anchor, so the offset relative to the TOC anchor is the displacement.

An undefined symbol cannot be used as a displacement for a D-form instruction. In other cases, its value is zero.

**TOC-Relative Relocatable Terms**

A term is TOC-relative relocatable (E_TREL) if it is a label contained within the TOC.

This type of term is relocatable since its value will change if the TOC is relocated.

A TOC-relative relocatable term is one of the following items:
- A label on a .tc pseudo-op
• A label defined within a csect that has TD or TC as its storage mapping class.

If this term is not used as a displacement for a D-form instruction, its value is its relocatable address, which is the sum of the offset relative to the TOC and the TOC anchor address. If it is used as a displacement for a D-form instruction, the assembler implicitly subtracts the TOC anchor address, so the offset relative to the TOC anchor is the displacement.

TOCOF Relocatable Terms
A term has TOCOF relocatable (E_TOCOF) type if it is the first operand of a .tocof pseudo-op.

This type of term has a value of zero. It cannot be used as a displacement for a D-form instruction. It cannot participate in any arithmetic operation.

Types and Values of Expressions
Expressions can have all the types that terms can have. An expression with only one term has the same type as its term. Expressions can also have restricted external relocatable (E_REXT) type, which a term cannot have because this type requires at least two terms.

Restricted External Relocatable Expressions
An expression has restricted external relocatable (E_REXT) type if it contains two relocatable terms that are defined in different control sections (terms not meeting the requirements for paired relocatable terms, as defined in "Expression Type of Combined Expressions" on page 43) and have opposite signs.

The following are examples of combinations of relocatable terms that produce an expression with restricted external relocatable type:
- \( <E_{\text{EXT}} > - <E_{\text{EXT}} > \)
- \( <E_{\text{REL}} > - <E_{\text{REL}} > \)
- \( <E_{\text{TREL}} > - <E_{\text{TREL}} > \)
- \( <E_{\text{EXT}} > - <E_{\text{REL}} > \)
- \( <E_{\text{REL}} > - <E_{\text{EXT}} > \)
- \( <E_{\text{TREL}} > - <E_{\text{REL}} > \)
- \( <E_{\text{EXT}} > - <E_{\text{TREL}} > \)
- \( <E_{\text{TREL}} > - <E_{\text{REL}} > \)

The value assigned to an expression of this type is based on the results of the assembler arithmetic evaluation of the values of its terms. When participating in an arithmetic operation, the value of a term is its relocatable address.

Combination Handling of Expressions
Terms within an expression can be combined with binary operators. Also, a term can begin with one or more unary operators. The assembler expression handler evaluates and determines the resultant expression type, value, and relocation table entries.

Expression Value Calculations
The following rules apply when calculating a value:
- If it is participating in an arithmetic operation, the value of an absolute term is its constant value, and the value of a relocatable term (E_EXT, E_REL, or E_TREL) is its relocatable address.
- If the resultant expression is used as a displacement in a D-form instruction, the assembler implicitly subtracts the containing csect address from the final result for expressions of type E_EXT or E_REL, or subtracts the TOC anchor address for expressions of type E_TREL. There is no implicit subtracting for expressions with E_ABS or E_REXT type.

Object File Relocation Table Entries of Expressions
The assembler applies the following rules when determining the requirements for object file relocation table entries for an expression.
• When an expression is used in a data definition, TOC entry definition, or a branch target address, it may require from zero to two relocation table entries (RLDs) depending on the resultant type of the expression.
  – E_ABS requires zero relocation entries.
  – E_REL requires one relocation entry, except that a dsect name or a dsect label does not require a relocation entry.
  – E_EXT requires one relocation entry
  – E_REXT requires two relocation entries
  – E_TREL requires one relocation entry
  – E_TOCOF requires one relocation entry

• When an expression is used as a displacement within a D-form instruction operand, only E_TREL and E_REXT expressions have relocation entries. They each require one relocation entry.

Expression Type of Combined Expressions

The assembler applies the following rules when determining the type of a combined expression.

**Combining Expressions with Group 1 Operators:** The following operators belong to group #1:
  • *, /, >, <, I, &, ^

Operators in group #1 have the following rules:
  • <E_ABS> <op1> <E_ABS> ==> E_ABS
  • Applying an operator in group #1 to any type of expression other than an absolute expression produces an error.

**Combining Expressions with Group 2 Operators:** The following operators belong to group # 2:
  • +, -

Operators in group # 2 have the following rules:
  • <E_ABS> <op2> <E_ABS> ==> E_ABS
  • <E_ABS> <op2> <E_REXT> ==> E_REXT
  • <E_REXT> <op2> <E_ABS> ==> E_REXT
  • <E_ABS> <op2> <E_TOCOF> ==> an error
  • <E_TOCOF> <op2> <E_ABS> ==> an error
  • <non E_ABS> <op2> <E_REXT> ==> an error
  • <E_REXT> <op2> <non E_ABS> ==> an error
  • <E_ABS> - <E_TREL> ==> an error
  • Unary + and - are treated the same as the binary operators with absolute value 0 (zero) as the left term.
  • Other situations where one of the terms is not an absolute expression require more complex rules.

The following definitions will be used in later discussion:

**paired relocatable terms** Have opposite signs and are defined in the same section. The value represented by paired relocatable terms is absolute. The result type for paired relocatable terms is E_ABS. Paired relocatable terms are not required to be contiguous in an expression. Two relocatable terms cannot be paired if they are not defined in the same section. A E_TREL term can be paired with another E_TREL term or E_EXT term, but cannot be paired with a E_REL term (because they will never be in the same section). A E_EXT or E_REL term can be paired with another E_EXT or E_REL term. A E_REXT term cannot be paired with any term.
opposite terms

Have opposite signs and point to the same symbol table entry. Any term can have its opposite term. The value represented by opposite terms is zero. The result type for opposite terms is almost identical to \( E_{ABS} \), except that a relocation table entry (RLD) with a type R_REF is generated when it is used for data definition. Opposite terms are not required to be contiguous in an expression.

The main difference between opposite terms and paired relocatable terms is that paired relocatable terms do not have to point to the same table entry, although they must be defined in the same section.

In the following example \( L1 \) and \(-L1\) are opposite terms ; and \( L1 \) and \(-L2\) are paired relocatable terms.

```
.file "f1.s"
.csect Dummy[PR]
L1: ai 10, 20, 30
L2: ai 11, 21, 30
br
   .csect A[RW]
   .long L1 - L1
   .long L1 - L2
```

The following table shows rules for determining the type of complex combined expressions:

<table>
<thead>
<tr>
<th>Type</th>
<th>Conditions for Expression to have Type</th>
<th>Relocation Table Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_{ABS} )</td>
<td>All the terms of the expression are paired relocatable terms, opposite terms, and absolute terms.</td>
<td>An RLD with type R_REF is generated for each opposite term.</td>
</tr>
<tr>
<td>( E_{REXT} )</td>
<td>The expression contains two unpaired relocatable terms with opposite signs in addition to all the paired relocatable terms, opposite terms, and absolute terms.</td>
<td>Two RLDs, one with a type of R_POS and one with a type of R_NEG, are generated for the unpaired relocatable terms. In addition, an RLD with a type of R_REF is generated for each opposite term.</td>
</tr>
<tr>
<td>( E_{REL}, E_{EXT} )</td>
<td>The expression contains only one unpaired ( E_{REL} ) or ( E_{REXT} ) term in addition to all the paired relocatable terms, opposite terms, and absolute terms.</td>
<td>If the expression is used in a data definition, one RLD with type R_POS or R_NEG will be generated. In addition, an RLD with type R_REF is generated for each opposite term.</td>
</tr>
<tr>
<td>( E_{TREL} )</td>
<td>The expression contains only one unpaired ( E_{TREL} ) term in addition to all the paired relocatable terms, opposite terms, and absolute terms.</td>
<td>If the expression is used as a displacement in a D-form instruction, one RLD with type R_TOC will be generated, otherwise one RLD with type R_POS or R_NEG will be generated. In addition, an RLD with type R_REF is generated for each opposite term.</td>
</tr>
<tr>
<td>Error</td>
<td>If the expression contains more than two unpaired relocatable terms, or it contains two unpaired relocatable terms with the same sign, an error is reported.</td>
<td></td>
</tr>
</tbody>
</table>
ba 16 + EL2 - L2 + L1  # Result is E_REL
1 10, 16+EL2-L2+L1(20)  # No RLD
.csect C[RW]
BL3: .long BL2 - B[PR]  # Result is E_ABS
    .long BL2 - (L1 - L1)  # Result is E_REL
    .long 14-(-EL2+BL1) + BL1 - (L2-L1)  # Result is E_REL
    .long 14 + EL2 - BL1 - L2 + L1  # Result is E_REL
    .long (B[PR] - A[PR]) + 32  # Result is E_REXT

Related Information
"Character Set" on page 27
"Reserved Words" on page 28
"Line Format" on page 28
"Statements" on page 29
"Symbols" on page 31
"Constants" on page 35
"Operators" on page 38

The `atof` subroutine.

".comm Pseudo-op" on page 471, ".csect Pseudo-op" on page 473, ".double Pseudo-op" on page 475,
Chapter 4. Addressing

The addressing articles discuss addressing modes and addressing considerations, including:

- "Absolute Addressing"
- "Absolute Immediate Addressing"
- "Relative Immediate Addressing" on page 48
- "Explicit-Based Addressing" on page 48
- "Implicit-Based Addressing" on page 50
- "Location Counter" on page 51

Absolute Addressing

An absolute address is represented by the contents of a register. This addressing mode is absolute in the sense that it is not specified relative to the current instruction address.

Both the Branch Conditional to Link Register instructions and the Branch Conditional to Count Register instructions use an absolute addressing mode. The target address is a specific register, not an input operand. The target register is the Link Register (LR) for the Branch Conditional to Link Register instructions. The target register is the Count Register (CR) for the Branch Conditional to Count Register instructions. These registers must be loaded prior to execution of the branch conditional to register instruction.

Related Information

- "Absolute Immediate Addressing."
- "Relative Immediate Addressing" on page 48.
- "Explicit-Based Addressing" on page 48.
- "Implicit-Based Addressing" on page 50.
- "Location Counter" on page 51.
- "Branch Processor" on page 19.

Absolute Immediate Addressing

An absolute immediate address is designated by immediate data. This addressing mode is absolute in the sense that it is not specified relative to the current instruction address.

For Branch and Branch Conditional instructions, an absolute immediate addressing mode is used if the Absolute Address bit (AA bit) is on.

The operand for the immediate data can be an absolute, relocatable, or external expression.
Related Information

“Absolute Addressing” on page 47.

“Relative Immediate Addressing.”

“Explicit-Based Addressing.”

“Implicit-Based Addressing” on page 50.

“Location Counter” on page 51.

“Branch Processor” on page 19.

“bcctr or bcc (Branch Conditional to Count Register) Instruction” on page 147, “bclr or bcr (Branch Conditional Link Register) Instruction” on page 149, “b (Branch) Instruction” on page 143, “bc (Branch Conditional) Instruction” on page 144.

Relative Immediate Addressing

Relative immediate addresses are specified as immediate data within the object code and are calculated relative to the current instruction location. All the instructions that use relative immediate addressing are branch instructions. These instructions have immediate data that is the displacement in full words from the current instruction location. At execution, the immediate data is sign extended, logically shifted to the left two bits, and added to the address of the branch instruction to calculate the branch target address. The immediate data must be a relocatable expression or an external expression.

Related Information

“Absolute Addressing” on page 47.

“Absolute Immediate Addressing” on page 47.

“Explicit-Based Addressing.”

“Implicit-Based Addressing” on page 50.

“Location Counter” on page 51.

“Branch Processor” on page 19.

“bcctr or bcc (Branch Conditional to Count Register) Instruction” on page 147, “bclr or bcr (Branch Conditional Link Register) Instruction” on page 149, “b (Branch) Instruction” on page 143, “bc (Branch Conditional) Instruction” on page 144.

“using Pseudo-op” on page 505, “.drop Pseudo-op” on page 476.

Explicit-Based Addressing

Explicit-based addresses are specified as a base register number, RA, and a displacement, D. The base register holds a base address. At run time, the processor adds the displacement to the contents of the base register to obtain the effective address. If an instruction does not have an operand form of D(RA), then the instruction cannot have an explicit-based address. Error 159 is reported if the D(RA) form is used for these instructions.
A displacement can be an absolute expression, a relocatable expression, a restricted external expression, or a TOC-relative expression. A displacement can be an external expression only if it is a csect (control section) name or the name of a common block specified defined by a \texttt{.comm} pseudo-op.

Notes:
1. An externalized label is still relocatable, so an externalized label can also be used as a displacement.
2. When a relocatable expression is used for the displacement, no RLD entry is generated, because only the offset from the label (that is, the relocatable expression) for the csect is used for the displacement.

Although programmers must use an absolute expression to specify the base register itself, the contents of the base register can be specified by an absolute, a relocatable, or an external expression. If the base register holds a relocatable value, the effective address is relocatable. If the base register holds an absolute value, the effective address is absolute. If the base register holds a value specified by an external expression, the type of the effective address is absolute if the expression is eventually defined as absolute, or relocatable if the expression is eventually defined as relocatable.

When using explicit-based addressing, remember that:
• GPR 0 cannot be used as a base register. Specifying 0 tells the assembler not to use a base register at all.
• Because $D$ occupies a maximum of 16 bits, a displacement must be in the range $-2^{15}$ to $(2^{15})-1$. Therefore, the difference between the base address and the address of the item to which reference is made must be less than $2^{15}$ bytes.

Note: $D$ and $RA$ are required for the $D(RA)$ form. The form $0(RA)$ or $D(0)$ may be used, but both the $D$ and $RA$ operands are required. There are two exceptions:
- When $D$ is an absolute expression,
- When $D$ is a restricted external expression.

If the $RA$ operand is missing in these two cases, $D(0)$ is assumed.

Related Information

"Absolute Addressing" on page 47.
"Absolute Immediate Addressing" on page 47.
"Relative Immediate Addressing" on page 48.
"Implicit-Based Addressing" on page 50.
"Location Counter" on page 51.
"Branch Processor" on page 19.
"bcctr or bcc (Branch Conditional to Count Register) Instruction" on page 147, "bclr or bcr (Branch Conditional Link Register) Instruction" on page 149, "b (Branch) Instruction" on page 143, "bc (Branch Conditional) Instruction" on page 144.
Implicit-Based Addressing

An implicit-based address is specified as an operand for an instruction by omitting the RA operand and writing the .using pseudo-op at some point before the instruction. After assembling the appropriate .using and .drop pseudo-ops, the assembler can determine which register to use as the base register. At run time, the processor computes the effective address just as if the base were explicitly specified in the instruction.

Implicit-based addresses can be relocatable or absolute, depending on the type of expression used to specify the contents of the RA operand at run time. Usually, the contents of the RA operand are specified with a relocatable expression, thus making a relocatable implicit-based address. In this case, when the object module produced by the assembler is relocated, only the contents of the base register RA will change. The displacement remains the same, so $D(RA)$ still points to the correct address after relocation.

A dsect is a reference control section that allows you to describe the layout of data in a storage area without actually reserving any storage. An implicit-based address can also be made by specifying the contents of RA with a dsect name or a a dsect label, thus associating a base with a dummy section. The value of the RA content is resolved at run time when the dsect is instantiated.

If the contents of the RA operand are specified with an absolute expression, an absolute implicit-based address is made. In this case, the contents of the RA will not change when the object module is relocated.

The assembler only supports relocatable implicit-based addressing.

Perform the following when using implicit-based addressing:

1. Write a .using statement to tell the assembler that one or more general-purpose registers (GPRs) will now be used as base registers.
2. In this .using statement, tell the assembler the value each base register will contain at execution. Until it encounters a .drop pseudo-op, the assembler will use this base register value to process all instructions that require a based address.
3. Load each base register with the previously specified value.

For implicit-based addressing the RA operand is always omitted, but the $D$ operand remains. The $D$ operand can be an absolute expression, a TOC-relative expression, a relocatable expression, or a restricted external expression.

Notes:

1. When the $D$ operand is an absolute expression or a restricted external expression, the assembler always converts it to $D(0)$ form, so the .using pseudo-op has no effect.
2. The .using and .drop pseudo-ops affect only based addresses.

See the “.using Pseudo-op” on page 505 for more information.
Location Counter

Each section of an assembler language program has a location counter used to assign storage addresses to your program’s statements. As the instructions of a source module are being assembled, the location counter keeps track of the current location in storage. You can use a $ (dollar sign) as an operand to an instruction to refer to the current value of the location counter.

Related Information

“Absolute Addressing” on page 47.
“Absolute Immediate Addressing” on page 47.
“Relative Immediate Addressing” on page 48.
“Explicit-Based Addressing” on page 48.
“Location Counter.”
“Branch Processor” on page 19.
“bcctr or bcc (Branch Conditional to Count Register) Instruction” on page 147, “bclr or bcr (Branch Conditional Link Register) Instruction” on page 149, “b (Branch) Instruction” on page 143, “bc (Branch Conditional) Instruction” on page 144.
“.using Pseudo-op” on page 505, “.drop Pseudo-op” on page 476.
Chapter 5. Assembling and Linking a Program

This section provides information on the following:

- "Assembling and Linking a Program"
- "Understanding Assembler Passes" on page 57
- "Interpreting an Assembler Listing" on page 59
- "Interpreting a Symbol Cross-Reference" on page 63
- "Subroutine Linkage Convention" on page 65
- "Understanding and Programming the TOC" on page 82
- "Running a Program" on page 87

Assembling and Linking a Program

Assembly language programs can be assembled with the `as` command or the `cc` command. The `ld` command or the `cc` command can be used to link assembled programs. This section discusses the following:

- "Assembling with the as Command"
- "Assembling and Linking with the cc Command" on page 56

Assembling with the as Command

The `as` command invokes the assembler. The syntax for the `as` command is as follows:

```
as [-a Mode ] [ -o ObjectFile ] [ -n Name ] [ -u ] [ -l [ListFile] ]
[ -W | -w ] [ -x [XCrossFile] ] [ -s [Listfile] ] [ -m ModeName ]
[ -Eoff]on ] [ -po]ff ]on ] [-i ] [-v ] [ File ]
```

The `as` command reads and assembles the file specified by the `File` parameter. By convention, this file has a suffix of `.s`. If no file is specified, the `as` command reads and assembles standard input. By default, the `as` command stores its output in a file named `a.out`. The output is stored in the XCOFF file format.

All flags for the `as` command are optional.

The `ld` command is used to link object files. See the `ld` command for more information.

The assembler respects the setting of the `OBJECT_MODE` environment variable. If neither `-a32` or `-a64` is used, the environment is examined for this variable. If the value of the variable is anything other than the values listed in the following table, an error message is generated and the assembler exits with a non-zero return code. The implied behavior corresponding to the valid settings are as follows:

<table>
<thead>
<tr>
<th><code>OBJECT_MODE</code></th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>32</code></td>
<td>Produce 32-bit object code. The default machine setting is <code>com</code>.</td>
</tr>
<tr>
<td><code>64</code></td>
<td>Produce 64-bit object code (XCOFF64 files). The default machine setting is <code>ppc64</code>.</td>
</tr>
<tr>
<td><code>32_64</code></td>
<td>Invalid.</td>
</tr>
<tr>
<td><code>anything else</code></td>
<td>Invalid.</td>
</tr>
</tbody>
</table>

as Command Flags

The following flags are recognized by the `as` command:

- `-a Mode` Specifies the mode in which the `as` command operates. By default, the `as` command operates in 32-bit mode, but the mode can be explicitly set by using the flag `-a32` for 32-bit mode operation or `-a64` for 64-bit mode operation.
-E[off/on] Specifies whether to report errors due to the v2.00 syntax (-Eon), or to ignore them (-Eoff). By default, v2.00 errors are ignored.

File Specifies the source file. If no file is specified, the source code is taken from standard input.

-i Specifies that branch prediction suffixes are to be encoded. By default, this option is not set. This option is ignored if the -p option is specified.

-li[ListFile] Produces an assembler listing. If you do not specify a file name, a default name is produced by replacing the suffix extension of the source file name with a .lst extension. (By convention, the source file suffix is a .s.) For example:

sourcefile.xyz produces a default name of:

sourcefile.lst

If the source code is from standard input and the -l flag is used without specifying an assembler-listing file name, the listing file name is a.lst.

-m ModeName Indicates the assembly mode. This flag has lower priority than the .machine pseudo-op.

If this flag is not used and no .machine pseudo-op is present in the source program, the default assembly mode is used. The default assembly mode has the POWER family/PowerPC intersection as the target environment, but treats all POWER family/PowerPC incompatibility errors (including instructions outside the POWER family/PowerPC intersection and invalid form errors) as instructional warnings.

If an assembly mode that is not valid is specified and no .machine pseudo-op is present in the source program, an error is reported and the default assembly mode is used for instruction validation in pass 1 of the assembler.

If the -m flag is used, the ModeName variable can specify one of the following values:

"" Explicitly specifies the default assembly mode which has the POWER family/PowerPC intersection as the target environment, but treats instructions outside the POWER family/PowerPC intersection and invalid form errors as instructional warnings. A space is required between -m and the null string argument (two double quotation marks).

com Specifies the POWER family/PowerPC intersection mode. A source program can contain only instructions that are common to both POWER family and PowerPC; any other instruction causes an error. Any instruction with an invalid form causes errors, terminates the assembly process, and results in no object code being generated.

Note: Certain POWER family instructions are supported by the PowerPC 601 RISC Microprocessor, but do not conform to the PowerPC architecture. These instructions cause errors when using the com assembly mode.

any Specifies the indiscriminate mode. The assembler generates object code for any recognized instruction, regardless of architecture. This mode is used primarily for operating system development and for testing and debugging purposes.

Note: All POWER family/PowerPC incompatibility errors are ignored when using the any assembly mode, and no warnings are generated.
-m ModeName

Specifies the PowerPC mode. A source program can contain only PowerPC instructions. Any other instruction causes an error.

Notes:
1. The PowerPC optional instructions are not implemented in every PowerPC processor and do not belong to the ppc mode. These instructions generate an error if they appear in a source program which is assembled using the ppc assembly mode.
2. Certain instructions conform to the PowerPC architecture, but are not supported by the PowerPC 601 RISC Microprocessor.

ppc64 Specifies the PowerPC 64-bit mode. A source program can contain 64-bit PowerPC instructions.

pwr Specifies the POWER family mode. A source program can contain only instructions for the POWER family implementation of the POWER family architecture.

pwr2(pwrx)
Specifies the POWER2 mode. A source program can contain only instructions for the POWER2 implementation of the POWER family architecture. pwrx is the preferred value. The alternate assembly mode value pwrx means the same thing as pwr2. Note: The POWER family implementation instruction set is a subset of the POWER2 implementation instruction set.

pwr5 Specifies the POWER5 mode. A source program can contain only instructions compatible with the POWER5 processor.

601 Specifies the PowerPC 601 RISC Microprocessor mode. A source program can contain only instructions for the PowerPC 601 RISC Microprocessor. Note: The PowerPC 601 RISC Microprocessor design was completed before the PowerPC architecture. Therefore, some PowerPC instructions may not be supported by the PowerPC 601 RISC Microprocessor.

Attention: It is recommended that the 601 assembly mode not be used for applications that are intended to be portable to future PowerPC systems. The com or ppc assembly mode should be used for such applications.

The PowerPC 601 RISC Microprocessor implements the PowerPC architecture plus some POWER family instructions which are not included in the PowerPC architecture. This allows existing POWER family applications to run with acceptable performance on PowerPC systems. Future PowerPC systems will not have this feature. The 601 assembly mode may result in applications that will not run on existing POWER family systems and that may not have acceptable performance on future PowerPC systems, because the 601 assembly mode permits the use of all the instructions provided by the PowerPC 601 RISC Microprocessor.

603 Specifies the PowerPC 603 RISC Microprocessor mode. A source program can contain only instructions for the PowerPC 603 RISC Microprocessor.

604 Specifies the PowerPC 604 RISC Microprocessor mode. A source program can contain only instructions for the PowerPC 604 RISC Microprocessor.

970 Specifies the PPC970 mode. A source program can contain only instructions compatible with the PPC970 processor.

A35 Specifies the A35 mode. A source program can contain only instructions for the A35.

-n Name
Specifies the name that appears in the header of the assembler listing. By default, the header contains the name of the assembler source file.

-o ObjectFile
Writes the output of the assembly process to the specified file instead of to the a.out file.

-p[off/on]
Specifies whether to use the v2.00 branch prediction (-pon) or pre-v2.00 branch prediction (-poff). By default, pre-v2.00 branch prediction is used.
-s[ListFile]
Indicates whether or not a mnemonics cross-reference for POWER family and PowerPC is included in the assembler listing. If this flag is omitted, no mnemonics cross-reference is produced. If this flag is used, the assembler listing will have POWER family mnemonics if the source contains PowerPC mnemonics, and will have PowerPC mnemonics if the source contains POWER family mnemonics.

The mnemonics cross-reference is restricted to instructions that have different mnemonics in POWER family and PowerPC, but that have the same op code, function, and input operand format.

Because the -s flag is used to change the assembler-listing format, it implies the -l flag. If both option flags are used and different assembler-listing file names (specified by the ListFile variable) are given, the listing file name specified by the ListFile variable used with the -l flag is used. If an assembler-listing file name is not specified with either the -l or -s flag, a default assembler listing file name is produced by replacing the suffix extension of the source file name with a .lst extension.

-u
Accepts an undefined symbol as an extern so that an error message is not displayed. Otherwise, undefined symbols are flagged with error messages.

-v
Displays the version number of this command.

-W
Turns off all warning message reporting, including the instructional warning messages (the POWER family and PowerPC incompatibility warnings).

-w
Turns on warning message reporting, including reporting of instructional warning messages (the POWER family and PowerPC incompatibility warnings).

Note: When neither -W nor -w is specified, the instructional warnings are reported, but other warnings are suppressed.

-x[XCrossFile]
Produces cross-reference output. If you do not specify a file name, a default name is produced by replacing the suffix extension of the source file name with an .xref extension. By convention, the suffix is a .s. For example:

sourcefile.xyz

produces a default name of:

sourcefile.xref

Note: The assembler does not generate an object file when the -x flag is used.

Assembling and Linking with the cc Command
The cc command can be used to assemble and link an assembly source program. The following example links object files compiled or assembled with the cc command:

cc pgm.o subs1.o subs2.o

When the cc command is used to link object files, the object files should have the suffix of .o as in the previous example.

When the cc command is used to assemble and link source files, any assembler source files must have the suffix of .s. The cc command invokes the assembler for any files having this suffix. Option flags for the as command can be directed to the assembler through the cc command. The syntax is:

-Wa,Option1,Option2,...

The following example invokes the assembler to assemble the source program using the com assembly mode, and produces an assembler listing and an object file:

cc -c -Wa,-mcom,-l file.s

The cc command invokes the assembler and then continues processing normally. Therefore:

cc -Wa,-l,-oXfile.o file.s
will fail because the object file produced by the assembler is named Xfile.o, but the linkage editor \( \text{ld} \) command invoked by the \text{cc} command searches for file.o.

If no option flag is specified on the command line, the \text{cc} command uses the compiler, assembler, and link options, as well as the necessary support libraries defined in the \text{xlc.cfg} configuration file.

\textbf{Note:} Some option flags defined in the assembler and the linkage editor use the same letters. Therefore, if the \text{xlc.cfg} configuration file is used to define the assembler options (\text{asopt}) and the link-editor options (\text{ldopt}), duplicate letters should not occur in \text{asopt} and \text{ldopt} because the \text{cc} command is unable to distinguish the duplicate letters.

For more information on the option flags passed to the \text{cc} command, see the \text{cc} command.

\textbf{Related Information}

"Understanding Assembler Passes."

"Interpreting an Assembler Listing" on page 59.

"Interpreting a Symbol Cross-Reference" on page 63.

"Subroutine Linkage Convention" on page 65.

"Understanding and Programming the TOC" on page 82.

"Running a Program" on page 87.

The \text{as} command, and the \text{ld} command.

\textbf{Understanding Assembler Passes}

When you enter the \text{as} command, the assembler makes two passes over the source program.

\textbf{First Pass}

On the first pass, the assembler performs the following tasks:

- Checks to see if the instructions are legal in the current assembly mode.
- Allocates space for instructions and storage areas you request.
- Fills in the values of constants, where possible.
- Builds a symbol table, also called a cross-reference table, and makes an entry in this table for every symbol it encounters in the label field of a statement.

The assembler reads one line of the source file at a time. If this source statement has a valid symbol in the label field, the assembler ensures that the symbol has not already been used as a label. If this is the first time the symbol has been used as a label, the assembler adds the label to the symbol table and assigns the value of the current location counter to the symbol. If the symbol has already been used as a label, the assembler returns the error message \text{Redefinition of symbol} and reassigns the symbol value.

Next, the assembler examines the instruction’s mnemonic. If the mnemonic is for a machine instruction that is legal for the current assembly mode, the assembler determines the format of the instruction (for example, XO format). The assembler then allocates the number of bytes necessary to hold the machine code for the instruction. The contents of the location counter are incremented by this number of bytes.

When the assembler encounters a comment (preceded by a \# (pound sign)) or an end-of-line character, the assembler starts scanning the next instruction statement. The assembler keeps scanning statements and building its symbol table until there are no more statements to read.
At the end of the first pass, all the necessary space has been allocated and each symbol defined in the program has been associated with a location counter value in the symbol table. When there are no more source statements to read, the second pass starts at the beginning of the program.

**Note:** If an error is found in the first pass, the assembly process terminates and does not continue to the second pass. If this occurs, the assembler listing only contains errors and warnings generated during the first pass of the assembler.

**Second Pass**

On the second pass, the assembler:

- Examines the operands for symbolic references to storage locations and resolves these symbolic references using information in the symbol table.
- Ensures that no instructions contain an invalid instruction form.
- Translates source statements into machine code and constants, thus filling the allocated space with object code.
- Produces a file containing error messages, if any have occurred.

At the beginning of the second pass, the assembler scans each source statement a second time. As the assembler translates each instruction, it increments the value contained in the location counter.

If a particular symbol appears in the source code, but is not found in the symbol table, then the symbol was never defined. That is, the assembler did not encounter the symbol in the label field of any of the statements scanned during the first pass, or the symbol was never the subject of a `.comm`, `.csect`, `.lcomm`, `.sect`, or `.set` pseudo-op.

This could be either a deliberate external reference or a programmer error, such as misspelling a symbol name. The assembler indicates an error. All external references must appear in a `.extern` or `.globl` statement.

The assembler logs errors such as incorrect data alignment. However, many alignment problems are indicated by statements that do not halt assembly. The `-w` flag must be used to display these warning messages.

After the programmer corrects assembly errors, the program is ready to be linked.

**Note:** If only warnings are generated in the first pass, the assembly process continues to the second pass. The assembler listing contains errors and warnings generated during the second pass of the assembler. Any warnings generated in the first pass do not appear in the assembler listing.

**Related Information**

- "Assembling and Linking a Program" on page 53.
- "Interpreting an Assembler Listing" on page 59.
- "Interpreting a Symbol Cross-Reference" on page 63.
- "Subroutine Linkage Convention" on page 65.
- "Understanding and Programming the TOC" on page 82.
- "Running a Program" on page 87.

The `as` command.
Interpreting an Assembler Listing

The `-l` flag of the `as` command produces a listing of an assembler language file.

Assume that a programmer wants to display the words "hello, world." The C program would appear as follows:

```c
main ( )
{
    printf ("hello, world\n");
}
```

Assembling the `hello.s` file with the following command:

```
as -l hello.s
```

produces an output file named `hello.lst`. The complete assembler listing for `hello.lst` is as follows:

```
V4.0 01/25/1994
---
File# Line# Mode Name Loc Ctr Object Code Source
0 1 | # C source code
0 2 | # hello()
0 3 | #}
0 4 | # printf("hello,world\n");
0 5 | #}
0 6 | # Compile as follows:
0 7 | # cc -o helloworld hello.s
0 8 | # Static data entry in
0 9 | .file "hello.s"
10 | #T(able)O(f)C(ontents)
11 | .file hello.s
12 | #Set routine stack variables
13 | #Values are specific to
14 | #the current routine and can
15 | #vary from routine to routine
16 | .data 00000000 00000040 .globl main[ds]
17 | .toc
18 | .globl main[ds]
19 | .csect main[ds]
20 | .csect main[ds]
21 | .long .main[PR]
22 | .long TOC[tc0]
23 | .long 0
24 | .csect main[ds]
25 | .csect main[ds]
26 | .long .main[PR]
27 | .globl .main[PR]
28 | .globl .main[PR]
29 | .set argarea, 32
30 | .set linkarea, 24
31 | .set locstkarea, 0
32 | .set ngprs, 1
33 | .set nfprs, 0
34 | .set szdsa, 8*nfprs+4*ngprs+linkarea+argarea+locstkarea
35 | .csect .main[
36 | #Main routine
37 | .main[PR]
38 | #PROLOG: Called Routines
```

Chapter 5. Assembling and Linking a Program  59
# Responsibilities

Get link reg.

Not required to Get/Save CR

because current routine does not alter it.

Not required to Save FPR's

14-31 because current routine does not alter them.

Save GPR 31.

Save LR if non-leaf routine.

Decrement stack ptr and save back chain.

Program body

Load static data address

Line 3, file hello.c

Load address of data string

This is a parameter to printf()

Call 3, helloworld(14)

Call printf function

15, 15, 15

EPILOG: Return Sequence

Get saved LR.

Routine did not save CR.

Restore of CR not necessary.

Restore stack ptr

Restore GPR 31.

Routine did not save FPR's.

Restore of FPR's not necessary.

Move return address
to Link Register.

Return to address held in Link Register.

External variables

.extern.printf[PR]

Data

String data placed in

static csect data[rw]

csect data[rw]

.align2

_helloworld:
The first line of the assembler listing gives two pieces of information:
- Name of the source file (in this case, **hello.s**)
- Date the listing file was created

The assembler listing contains several columns. The column headings are:

<table>
<thead>
<tr>
<th>File#</th>
<th>Line#</th>
<th>Mode</th>
<th>Name</th>
<th>Loc Ctr</th>
<th>Object Code</th>
<th>PowerPC</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>114</td>
<td>COM</td>
<td>data</td>
<td>00000000</td>
<td>68656c6c</td>
<td>.byte</td>
<td>0x68,0x65,0x6c,0x6c</td>
</tr>
<tr>
<td>0</td>
<td>115</td>
<td>COM</td>
<td>data</td>
<td>00000004</td>
<td>6f2c776f</td>
<td>.byte</td>
<td>0x6f,0x2c,0x77,0x6f</td>
</tr>
<tr>
<td>0</td>
<td>116</td>
<td>COM</td>
<td>data</td>
<td>00000008</td>
<td>72bc640a</td>
<td>.byte</td>
<td>0x72,0x6c,0x64,0xa,0x0</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>COM</td>
<td>data</td>
<td>0000000c</td>
<td>00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If the -s option flag is used on the command line, the assembler listing contains mnemonic cross-reference information.

If the assembly mode is in the PowerPC category (**com**, **ppc**, or **601**), one column heading is PowerPC. This column contains the PowerPC mnemonic for each instance where the POWER family mnemonic is used in the source program. The **any** assembly mode does not belong to any category, but is treated as though in the PowerPC category.

If the assembly mode is in the POWER family category (**pwr** or **pwr2**), one column heading is POWER family. This column contains the POWER family mnemonic for each instance where the PowerPC mnemonic is used in the source program.

The following assembler listing uses the **com** assembly mode. The source program uses POWER family mnemonics. The assembler listing has a PowerPC mnemonic cross-reference.
<table>
<thead>
<tr>
<th>File#</th>
<th>Line#</th>
<th>Mode</th>
<th>Name</th>
<th>Loc Ctr</th>
<th>Object Code</th>
<th>POWER</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000000</td>
<td>8025000c</td>
<td>1</td>
<td>l</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000004</td>
<td>b8650018</td>
<td>1m</td>
<td>lmw</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000008</td>
<td>b0e50040</td>
<td>sth</td>
<td>7,d8</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>PWR</td>
<td>dfmt</td>
<td>0000000c</td>
<td>80230020</td>
<td>1</td>
<td>l</td>
</tr>
<tr>
<td>0</td>
<td>9</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000010</td>
<td>30220003</td>
<td>ai</td>
<td>addic</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000014</td>
<td>0cd78300</td>
<td>ti</td>
<td>twi</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000018</td>
<td>2c070af0</td>
<td>cmpi</td>
<td>0,7,2800</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>PWR</td>
<td>dfmt</td>
<td>0000001c</td>
<td>2c070af0</td>
<td>cmpi</td>
<td>0,7,2800</td>
</tr>
<tr>
<td>0</td>
<td>13</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000020</td>
<td>30220003</td>
<td>si</td>
<td>subic</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000024</td>
<td>34220003</td>
<td>si.</td>
<td>subic.</td>
</tr>
<tr>
<td>0</td>
<td>15</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000028</td>
<td>703e0ff</td>
<td>andil.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>PWR</td>
<td>dfmt</td>
<td>0000002c</td>
<td>2b9401f4</td>
<td>cmpi</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>17</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000030</td>
<td>0c2501a4</td>
<td>tlgli</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>18</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000034</td>
<td>34220003</td>
<td>ai.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>19</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000038</td>
<td>2c9ff380</td>
<td>cmpi</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>20</td>
<td>PWR</td>
<td>dfmt</td>
<td>0000003c</td>
<td>2810f0c0</td>
<td>cmpli</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>21</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000040</td>
<td>8ba5000c</td>
<td>lbz</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>22</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000044</td>
<td>85e5000c</td>
<td>lwzu</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>23</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000048</td>
<td>1df5ece0</td>
<td>muli</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>24</td>
<td>PWR</td>
<td>dfmt</td>
<td>0000004c</td>
<td>62af0140</td>
<td>ori</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000050</td>
<td>91e5000c</td>
<td>stw</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>26</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000054</td>
<td>bde5000c</td>
<td>stmw</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>27</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000058</td>
<td>95e5000c</td>
<td>stwu</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>28</td>
<td>PWR</td>
<td>dfmt</td>
<td>0000005c</td>
<td>69ef0960</td>
<td>xorl</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>29</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000060</td>
<td>6d8c0960</td>
<td>xoris</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>30</td>
<td>PWR</td>
<td>dfmt</td>
<td>00000064</td>
<td>3a9eff38</td>
<td>addi</td>
<td></td>
</tr>
</tbody>
</table>

The following assembler listing uses the **PWR** assembly mode. The source program uses PowerPC mnemonics. The assembler listing has a POWER family mnemonic cross-reference.

```
../dfmt | long 0,0
.
..long 3,4,5 # d1 = 0xC = 12
..long data # d0 = 0x18 = 24
..long data2: .space 36
..long 9184 # d8 = 0x40 = 64
..long 0xFFFFFFF # d9 = 0x44
```

**Assembler Language Reference**
## Related Information

- "Assembling and Linking a Program" on page 53.
- "Understanding Assembler Passes" on page 57.
- "Interpreting a Symbol Cross-Reference."
- "Subroutine Linkage Convention" on page 65.
- "Understanding and Programming the TOC" on page 82.
- "Running a Program" on page 87.

The [as command](#).

## Interpreting a Symbol Cross-Reference

The following is an example of the symbol cross-reference for the hello.s assembly program:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>File</th>
<th>CSECT</th>
<th>Line</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>.main</td>
<td>hello.s</td>
<td>--</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>.main</td>
<td>hello.s</td>
<td>.main</td>
<td>28</td>
<td>*</td>
</tr>
<tr>
<td>.main</td>
<td>hello.s</td>
<td>--</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>.main</td>
<td>hello.s</td>
<td>.main</td>
<td>43</td>
<td>*</td>
</tr>
<tr>
<td>printf</td>
<td>hello.s</td>
<td>--</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>printf</td>
<td>hello.s</td>
<td>--</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>T.data</td>
<td>hello.s</td>
<td>data</td>
<td>17</td>
<td>*</td>
</tr>
<tr>
<td>T.data</td>
<td>hello.s</td>
<td>data</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>T.hello</td>
<td>hello.s</td>
<td>.main</td>
<td>28</td>
<td>*</td>
</tr>
<tr>
<td>TOC</td>
<td>hello.s</td>
<td>TOC</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>_helloworld</td>
<td>hello.s</td>
<td>--</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>_helloworld</td>
<td>hello.s</td>
<td>data</td>
<td>113</td>
<td>*</td>
</tr>
<tr>
<td>argarea</td>
<td>hello.s</td>
<td>--</td>
<td>35</td>
<td>*</td>
</tr>
<tr>
<td>argarea</td>
<td>hello.s</td>
<td>--</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>hello.s</td>
<td>--</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>hello.s</td>
<td>data</td>
<td>17</td>
<td>*</td>
</tr>
<tr>
<td>data</td>
<td>hello.s</td>
<td>data</td>
<td>111</td>
<td>*</td>
</tr>
<tr>
<td>linkarea</td>
<td>hello.s</td>
<td>--</td>
<td>36</td>
<td>*</td>
</tr>
</tbody>
</table>

Chapter 5. Assembling and Linking a Program
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Source File</th>
<th>Csect</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>linkarea</td>
<td>hello.s</td>
<td>--</td>
<td>40</td>
</tr>
<tr>
<td>locstckarea</td>
<td>hello.s</td>
<td>--</td>
<td>37 *</td>
</tr>
<tr>
<td>locstckarea</td>
<td>hello.s</td>
<td>--</td>
<td>40</td>
</tr>
<tr>
<td>main</td>
<td>hello.s</td>
<td>--</td>
<td>18</td>
</tr>
<tr>
<td>main</td>
<td>hello.s</td>
<td>main</td>
<td>21 *</td>
</tr>
<tr>
<td>main</td>
<td>hello.s</td>
<td>main</td>
<td>28</td>
</tr>
<tr>
<td>nfprs</td>
<td>hello.s</td>
<td>--</td>
<td>39 *</td>
</tr>
<tr>
<td>nfprs</td>
<td>hello.s</td>
<td>--</td>
<td>40</td>
</tr>
<tr>
<td>nfprs</td>
<td>hello.s</td>
<td>--</td>
<td>59</td>
</tr>
<tr>
<td>nfprs</td>
<td>hello.s</td>
<td>--</td>
<td>90</td>
</tr>
<tr>
<td>ngprs</td>
<td>hello.s</td>
<td>--</td>
<td>38 *</td>
</tr>
<tr>
<td>ngprs</td>
<td>hello.s</td>
<td>--</td>
<td>40</td>
</tr>
<tr>
<td>ngprs</td>
<td>hello.s</td>
<td>--</td>
<td>59</td>
</tr>
<tr>
<td>ngprs</td>
<td>hello.s</td>
<td>--</td>
<td>90</td>
</tr>
<tr>
<td>szdsa</td>
<td>hello.s</td>
<td>--</td>
<td>40 *</td>
</tr>
<tr>
<td>szdsa</td>
<td>hello.s</td>
<td>--</td>
<td>64</td>
</tr>
<tr>
<td>szdsa</td>
<td>hello.s</td>
<td>--</td>
<td>82</td>
</tr>
<tr>
<td>szdsa</td>
<td>hello.s</td>
<td>--</td>
<td>88</td>
</tr>
</tbody>
</table>

The first column lists the symbol names that appear in the source program. The second column lists the source file name in which the symbols are located. The third column lists the csect names in which the symbols are defined or located.

In the column listing the csect names, a — (double dash) means one of the following:
- The symbol’s csect has not been defined yet. In the example, the first and third `.main (.main[PR])` is defined through line 42.
- The symbol is an external symbol. In the example, `.printf` is an external symbol and, therefore, is not associated with any csect.
- The symbol to be defined is a symbolic constant. When the `.set` pseudo-op is used to define a symbol, the symbol is a symbolic constant and does not have a csect associated with it. In the example, `argarea`, `linkarea`, `locstckarea`, `nfprs`, `ngprs`, and `szdsa` are symbolic constants.

The fourth column lists the line number in which the symbol is located. An * (asterisk) after the line number indicates that the symbol is defined in this line. If there is no asterisk after the line number, the symbol is referenced in the line.

### Related Information
- "Assembling and Linking a Program" on page 53.
- "Understanding Assembler Passes" on page 57.
- "Interpreting an Assembler Listing" on page 59.
- "Subroutine Linkage Convention" on page 65.
- "Understanding and Programming the TOC" on page 82.
- "Running a Program" on page 87.
- Chapter 7, "Migrating Source Programs," on page 113.

The `as` command.
Subroutine Linkage Convention

This article discusses the following:
- “Linkage Convention Overview”
- “Calling Routine’s Responsibilities” on page 77
- “Called Routine’s Responsibilities” on page 77
- “Using Milicode Routines” on page 80

Linkage Convention Overview

The subroutine linkage convention describes the machine state at subroutine entry and exit. When followed, this scheme allows routines compiled separately in the same or different languages to be linked and executed when called.

The linkage convention allows for parameter passing and return values to be in floating-point registers (FPRs), general-purpose registers (GPRs), or both.

Object Mode Considerations

The following discussion applies to both 32-bit mode and 64-bit mode with the following notes:
- General purpose registers in 64-bit mode are 64 bits wide (double-word). This implies that space usage of the stack increases by a factor of two for register storage. Wherever, below, the term word is used, assume (unless otherwise stated) that the size of the object in question is 1 word in 32-bit mode, and 2 words (a double-word) in 64-bit mode.
- The offsets shown in the runtime stack figure should be doubled for 64-bit mode. In 32-bit mode, the stack as shown requires 56 bytes:
  - 1 word for each of the 6 registers CR, LR, compiler-reserved, linker-reserved, and saved-TOC.
  - 8 words for the 8 volatile registers.

This totals 14 words, or 56 bytes. In 64-bit mode, each field is twice as large (a double-word), thus requiring 28 words, or 112 bytes.
- Floating point registers are saved in the same format in both modes. The storage requirements are the same.
- Stack pointer alignment requirements remain the same for both modes.
- The GPR save routine listed below illustrates the methodology for saving registers in 32-bit mode. For 64-bit mode, the offsets from GPR1, the stack pointer register, would be twice the values shown. Additionally, the load instruction used would be \texttt{ld} and the store instruction would be \texttt{stdu}.

Register Usage and Conventions

The PowerPC 32-bit architecture has 32 GPRs and 32 FPRs. Each GPR is 32 bits wide, and each FPR is 64 bits wide. There are also special registers for branching, exception handling, and other purposes. The General-Purpose Register Convention table shows how GPRs are used.

<table>
<thead>
<tr>
<th>Register</th>
<th>Status</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR0</td>
<td>volatile</td>
<td>In function prologs.</td>
</tr>
<tr>
<td>GPR1</td>
<td>dedicated</td>
<td>Stack pointer.</td>
</tr>
<tr>
<td>GPR2</td>
<td>dedicated</td>
<td>Table of Contents (TOC) pointer.</td>
</tr>
<tr>
<td>GPR3</td>
<td>volatile</td>
<td>First word of a function’s argument list; first word of a scalar function return.</td>
</tr>
<tr>
<td>GPR4</td>
<td>volatile</td>
<td>Second word of a function’s argument list; second word of a scalar function return.</td>
</tr>
<tr>
<td>GPR5</td>
<td>volatile</td>
<td>Third word of a function’s argument list.</td>
</tr>
</tbody>
</table>
Table 2. General-Purpose Register Conventions (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Status</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR6</td>
<td>volatile</td>
<td>Fourth word of a function’s argument list.</td>
</tr>
<tr>
<td>GPR7</td>
<td>volatile</td>
<td>Fifth word of a function’s argument list.</td>
</tr>
<tr>
<td>GPR8</td>
<td>volatile</td>
<td>Sixth word of a function’s argument list.</td>
</tr>
<tr>
<td>GPR9</td>
<td>volatile</td>
<td>Seventh word of a function’s argument list.</td>
</tr>
<tr>
<td>GPR10</td>
<td>volatile</td>
<td>Eighth word of a function’s argument list.</td>
</tr>
<tr>
<td>GPR11</td>
<td>volatile</td>
<td>In calls by pointer and as an environment pointer for languages that require it (for example, PASCAL).</td>
</tr>
<tr>
<td>GPR12</td>
<td>volatile</td>
<td>For special exception handling required by certain languages and in glink code.</td>
</tr>
<tr>
<td>GPR13</td>
<td>reserved</td>
<td>Reserved under 64-bit environment; not restored across system calls.</td>
</tr>
<tr>
<td>GPR14-GPR31</td>
<td>nonvolatile</td>
<td>These registers must be preserved across a function call.</td>
</tr>
</tbody>
</table>

The preferred method of using GPRs is to use the volatile registers first. Next, use the nonvolatile registers in descending order, starting with GPR31 and proceeding down to GPR14. GPR1 and GPR2 must be dedicated as stack and Table of Contents (TOC) area pointers, respectively. GPR1 and GPR2 must appear to be saved across a call, and must have the same values at return as when the call was made.

Volatile registers are scratch registers presumed to be destroyed across a call and are, therefore, not saved by the callee. Volatile registers are also used for specific purposes as shown in the previous table. Nonvolatile and dedicated registers are required to be saved and restored if altered and, thus, are guaranteed to retain their values across a function call.

The Floating-Point Register Conventions table shows how the FPRs are used.

Table 3. Floating-Point Register Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Status</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPR0</td>
<td>volatile</td>
<td>As a scratch register.</td>
</tr>
<tr>
<td>FPR1</td>
<td>volatile</td>
<td>First floating-point parameter; first 8 bytes of a floating-point scalar return.</td>
</tr>
<tr>
<td>FPR2</td>
<td>volatile</td>
<td>Second floating-point parameter; second 8 bytes of a floating-point scalar return.</td>
</tr>
<tr>
<td>FPR3</td>
<td>volatile</td>
<td>Third floating-point parameter; third 8 bytes of a floating-point scalar return.</td>
</tr>
<tr>
<td>FPR4</td>
<td>volatile</td>
<td>Fourth floating-point parameter; fourth 8 bytes of a floating-point scalar return.</td>
</tr>
<tr>
<td>FPR5</td>
<td>volatile</td>
<td>Fifth floating-point parameter.</td>
</tr>
<tr>
<td>FPR6</td>
<td>volatile</td>
<td>Sixth floating-point parameter.</td>
</tr>
<tr>
<td>FPR7</td>
<td>volatile</td>
<td>Seventh floating-point parameter.</td>
</tr>
<tr>
<td>FPR8</td>
<td>volatile</td>
<td>Eighth floating-point parameter.</td>
</tr>
<tr>
<td>FPR9</td>
<td>volatile</td>
<td>Ninth floating-point parameter.</td>
</tr>
<tr>
<td>FPR10</td>
<td>volatile</td>
<td>Tenth floating-point parameter.</td>
</tr>
<tr>
<td>FPR11</td>
<td>volatile</td>
<td>Eleventh floating-point parameter.</td>
</tr>
<tr>
<td>FPR12</td>
<td>volatile</td>
<td>Twelfth floating-point parameter.</td>
</tr>
<tr>
<td>FPR13</td>
<td>volatile</td>
<td>Thirteenth floating-point parameter.</td>
</tr>
<tr>
<td>FPR14-FPR31</td>
<td>nonvolatile</td>
<td>If modified, must be preserved across a call.</td>
</tr>
</tbody>
</table>
The preferred method of using FPRs is to use the volatile registers first. Next, the nonvolatile registers are used in descending order, starting with FPR31 and proceeding down to FPR14.

Only scalars are returned in multiple registers. The number of registers required depends on the size and type of the scalar. For floating-point values, the following results occur:

- A 128-bit floating-point value returns the high-order 64 bits in FPR1 and the low-order 64 bits in FPR2.
- An 8-byte or 16-byte complex value returns the real part in FPR1 and the imaginary part in FPR2.
- A 32-byte complex value returns the real part as a 128-bit floating-point value in FPR1 and FPR2, with the high-order 64 bits in FPR1 and the low-order 64 bits in FPR2. The imaginary part of a 32-byte complex value returns the high-order 64 bits in FPR3 and the low-order 64 bits in FPR4.

**Special Registers in the PowerPC**

The Special-Purpose Register Conventions table shows the PowerPC special purpose registers (SPRs). These are the only SPRs for which there is a register convention.

<table>
<thead>
<tr>
<th>Register or Register Field</th>
<th>Status</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>volatile</td>
<td>Used as a branch target address or holds a return address.</td>
</tr>
<tr>
<td>CTR</td>
<td>volatile</td>
<td>Used for loop count decrement and branching.</td>
</tr>
<tr>
<td>XER</td>
<td>volatile</td>
<td>Fixed-point exception register.</td>
</tr>
<tr>
<td>FPSCR</td>
<td>volatile</td>
<td>Floating-point exception register.</td>
</tr>
<tr>
<td>CR0, CR1</td>
<td>volatile</td>
<td>Condition-register bits.</td>
</tr>
<tr>
<td>CR5, CR6, CR7</td>
<td>volatile</td>
<td>Condition-register bits.</td>
</tr>
</tbody>
</table>

Routines that alter CR2, CR3, and CR4 must save and restore at least these fields of the CR. Use of other CR fields does not require saving or restoring.

**Runtime Process Stack**

The stack format convention is designed to enhance the efficiency of the following:

- Prolog and epilog function usage
- Parameter passing
- Shared library support

The Runtime Stack figure illustrates the runtime stack. It shows the stack after the sender function calls the catcher function, but before the catcher function calls another function. This figure is based on the assumption that the catcher function will call another function. Therefore, the catcher function requires another link area (as described in the stack layout). $PW_n$ refers to the $n$th word of parameters that are passed.
Stack Layout: Only one register, referred to as the stack pointer (SP), is used for addressing the stack, and GPR1 is the dedicated stack pointer register. It grows from numerically higher storage addresses to numerically lower addresses.

The Runtime Stack figure illustrates what happens when the sender function calls the catcher function, and how the catcher function requires a stack frame of its own. When a function makes no calls and requires no local storage of its own, no stack frame is required and the SP is not altered.
Notes:
1. To reduce confusion, data being passed from the sender function (the caller) is referred to as arguments, and the same data being received by the catcher function (the callee) is referred to as parameters. The output argument area of sender is the same as the input parameter area of catcher.
2. The address value in the stack pointer must be quadword-aligned. (The address value must be a multiple of 16.)

Stack Areas: For convenience, the stack layout has been divided into eight areas numbered 1 to 8, starting from the bottom of the diagram (high address) to the top of the diagram (low address). The sender’s stack pointer is pointing to the top of area 3 when the call to the catcher function is made, which is also the same SP value that is used by the catcher function on entry to its prolog. The following is a description of the stack areas, starting from the bottom of the diagram (area 1) and moving up to the top (area 8):

- **Area 1: Sender’s Local Variable Area**
  Area 1 is the local variable area for the sender function, contains all local variables and temporary space required by this function.

- **Area 2: Sender’s Output Argument Area**
  Area 2 is the output argument area for the sender function. This area is at least eight words in size and must be doubleword-aligned. The first eight words are not used by the caller (the sender function) because their corresponding values are placed directly in the argument registers (GPR3:GPR10). The storage is reserved so that if the callee (the catcher function) takes the address of any of its parameters, the values passed in GPR3:GPR10 can be stored in their address locations (PW1:PW8, respectively). If the sender function is passing more than eight arguments to the catcher function, then it must reserve space for the excess parameters. The excess parameters must be stored as register images beyond the eight reserved words starting at offset 56 from the sender function’s SP value.

  **Note:** This area may also be used by language processors and is volatile across calls to other functions.

- **Area 3: Sender’s Link Area**
  Area 3 is the link area for the sender function. This area consists of six words and is at offset 0 from the sender function’s SP at the time the call to the catcher function is made. Certain fields in this area are used by the catcher function as part of its prolog code, those fields are marked in the Runtime Stack figure and are explained below.

  The first word is the back chain, the location where the sender function saved its caller’s SP value prior to modifying the SP. The second word (at offset 4) is where the catcher function can save the CR if it modifies any of the nonvolatile CR fields. The third word (offset 8) is where the catcher function can save the LR if the catcher function makes any calls.

  The fourth word is reserved for compilers, and the fifth word is used by binder-generated instructions. The last word in the link area (offset 20) is where the TOC area register (see “Understanding and Programming the TOC” on page 82 for description) is saved by the global linkage (glink) interface routine. This occurs when an out-of-module call is performed, such as when a shared library function is called.

- **Area 4: Catcher’s Floating-Point Registers Save Area**
  Area 4 is the floating-point register save area for the callee (the catcher function) and is doubleword-aligned. It represents the space needed to save all the nonvolatile FPRs used by the called program (the catcher function). The FPRs are saved immediately above the link area (at a lower address) at a negative displacement from the sender function’s SP. The size of this area varies from zero to a maximum of 144 bytes, depending on the number of FPRs being saved (maximum number is 18 FPRs * 8 bytes each).

- **Area 5: Catcher’s General-Purpose Registers Save Area**
  Area 5 is the general-purpose register save area for the catcher function and is at least word-aligned. It represents the space needed by the called program (the catcher function) to save all the nonvolatile GPRs. The GPRs are saved immediately above the FPR save area (at a lower address) at a negative
displacement from the `sender` function’s SP. The size of this area varies from zero to a maximum of 76 bytes, depending on the number of GPRs being saved (maximum number is 19 GPRs * 4 bytes each).

**Notes:**
1. A stackless leaf procedure makes no calls and requires no local variable area, but it may use nonvolatile GPRs and FPRs.
2. The save area consists of the FPR save area (4) and the GPR save area (5), which have a combined maximum size of 220 bytes. The stack floor of the currently executing function is located at 220 bytes less than the value in the SP. The area between the value in the SP and the stack floor is the maximum save area that a stackless leaf function may use without acquiring its own stack. Functions may use this area as temporary space which is volatile across calls to other functions. Execution elements such as interrupt handlers and binder-inserted code, which cannot be seen by compiled codes as calls, must not use this area.

The system-defined stack floor includes the maximum possible save area. The formula for the size of the save area is:

\[
18 \times 8 \\
+ 19 \times 4 \\
= 220
\]

**Area 6: Catcher’s Local Variable Area**

Area 6 is the local variable area for the `catcher` function and contains local variables and temporary space required by this function. The `catcher` function addresses this area using its own SP, which points to the top of area 8, as a base register.

**Area 7: Catcher’s Output Argument Area**

Area 7 is the output argument area for the `catcher` function and is at least eight words in size and must be doubleword-aligned. The first eight words are not used by the caller (the `catcher` function), because their corresponding values are placed directly in the argument registers (GPR3:GPR10). The storage is reserved so that if the `catcher` function’s callee takes the address of any of its parameters, then the values passed in GPR3:GPR10 can be stored in their address locations. If the `catcher` function is passing more than eight arguments to its callee (PW1:PW8, respectively), it must reserve space for the excess parameters. The excess parameters must be stored as register images beyond the eight reserved words starting at offset 56 from the `catcher` function’s SP value.

**Note:** This area can also be used by language processors and is volatile across calls to other functions.

**Area 8: Catcher’s Link Area**

Area 8 is the link area for the `catcher` function and contains the same fields as those in the `sender` function’s link area (area 3).

**Stack-Related System Standard**

All language processors and assemblers must maintain the stack-related system standard that the SP must be atomically updated by a single instruction. This ensures that there is no timing window where an interrupt that would result in the stack pointer being only partially updated can occur.

**Note:** The examples of program prologs and epilogs show the most efficient way to update the stack pointer.

**Prologs and Epilogs**

Prologs and epilogs may be used for functions, including setting the registers on function entry and restoring the registers on function exit.

No predetermined code sequences representing function prologs and epilogs are dictated. However, certain operations must be performed under certain conditions. The following diagram shows the stack frame layout.
A typical function's execution stack is:

- **Prolog action**
- **Body of function**
- **Epilog action**

The Prolog Actions and Epilog Actions tables show the conditions and actions required for prologs and epilogs.

**Table 5. Prolog Actions**

<table>
<thead>
<tr>
<th>If:</th>
<th>Then:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any nonvolatile FPRs (FPR14:FPR31) are used</td>
<td>Save them in the FPR save area (area 4 in the previous figure).</td>
</tr>
<tr>
<td>Any nonvolatile GPRs (GPR13:GPR31) are used</td>
<td>Save them in the GPR save area (area 5 in the previous figure).</td>
</tr>
<tr>
<td>LR is used for a nonleaf procedure</td>
<td>Save the LR at offset eight from the caller function SP.</td>
</tr>
</tbody>
</table>

*Figure 3. Stack Frame Layout*

A typical function’s execution stack is:

- **Prolog action**
- **Body of function**
- **Epilog action**
Table 5. Prolog Actions (continued)

<table>
<thead>
<tr>
<th>If:</th>
<th>Then:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any of the nonvolatile condition register (CR) fields are used.</td>
<td>Save the CR at offset four from the caller function SP.</td>
</tr>
<tr>
<td>A new stack frame is required</td>
<td>Get a stack frame and decrement the SP by the size of the frame padded (if necessary) to a multiple of 16 to acquire a new SP and save caller’s SP at offset 0 from the new SP.</td>
</tr>
</tbody>
</table>

**Note:** A leaf function that does not require stack space for local variables and temporaries can save its caller registers at a negative offset from the caller SP without actually acquiring a stack frame.

Table 6. Epilog Actions

<table>
<thead>
<tr>
<th>If:</th>
<th>Then:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any nonvolatile FPRs were saved</td>
<td>Restore the FPRs that were used.</td>
</tr>
<tr>
<td>Any nonvolatile GPRs were saved</td>
<td>Restore the GPRs that were saved.</td>
</tr>
<tr>
<td>The LR was altered because a nonleaf procedure was invoked</td>
<td>Restore LR.</td>
</tr>
<tr>
<td>The CR was altered</td>
<td>Restore CR.</td>
</tr>
<tr>
<td>A new stack was acquired</td>
<td>Restore the old SP to the value it had on entry (the caller's SP). Return to caller.</td>
</tr>
</tbody>
</table>

While the PowerPC architecture provides both load and store multiple instructions for GPRs, it discourages their use because their implementation on some machines may not be optimal. In fact, use of the load and store multiple instructions on some future implementations may be significantly slower than the equivalent series of single word loads or stores. However, saving many FPRs or GPRs with single load or store instructions in a function prolog or epilog leads to increased code size. For this reason, the system environment must provide routines that can be called from a function prolog and epilog that will do the saving and restoring of the FPRs and GPRs. The interface to these routines, their source code, and some prolog and epilog code sequences are provided.

As shown in the stack frame layout, the GPR save area is not at a fixed position from either the caller SP or the callee SP. The FPR save area starts at a fixed position, directly above the SP (lower address) on entry to that callee, but the position of the GPR save area depends on the number of FPRs saved. Thus, it is difficult to write a general-purpose GPR-saving function that uses fixed displacements from SP.

If the routine needs to save both GPRs and FPRs, use GPR12 as the pointer for saving and restoring GPRs. (GPR12 is a volatile register, but does not contain input parameters.) This results in the definition of multiple-register save and restore routines, each of which saves or restores \( m \) FPRs and \( n \) GPRs. This is achieved by executing an **bla** (Branch and Link Absolute) instruction to specially provided routines containing multiple entry points (one for each register number), starting from the lowest nonvolatile register.

**Notes:**

1. There are no entry points for saving and restoring GPR and FPR numbers greater than 29. It is more efficient to save a small number of registers in the prolog than it is to call the save and restore functions.
2. If the LR is not saved or restored in the following code segments, the language processor must perform the saving and restoring as appropriate.

Language processors must use a proprietary method to conserve the values of nonvolatile registers across a function call.
Three sets of save and restore routines must be made available by the system environment. These routines are:
- A pair of routines to save and restore GPRs when FPRs are not being saved and restored.
- A pair of routines to save and restore GPRs when FPRs are being saved and restored.
- A pair of routines to save and restore FPRs.

**Saving GPRs Only:** For a function that saves and restores \( n \) GPRs and no FPRs, the saving can be done using individual store and load instructions or by calling system-provided routines as shown in the following example:

**Note:** The number of registers being saved is \( n \). Sequences such as \(<32-n>\) in the following examples indicate the first register number to be saved and restored. All registers from \(<32-n>\) to 31, inclusive, are saved and restored.

```assembly
mflr r0 #move LR into GPR0
bla _savegpr0_<32-n> #branch and link to save GPRs
stwu r1,-<frame_size>(r1) #update SP and save caller's SP
... #frame_size is the size of the stack frame to be required

<save CR if necessary>
...
#body of function
...
<reload save CR if necessary>
...
<reload caller's SP into R1> #see note below
ba _restgpr0_<32-n> #restore GPRs and return
```

**Note:** The restoring of the calling function SP can be done by either adding the frame_size value to the current SP whenever frame_size is known, or by reloading it from offset 0 from the current SP. The first approach is more efficient, but not possible for functions that use the **alloca** subroutine to dynamically allocate stack space.

The following example shows a GPR save routine when FPRs are not saved:

```assembly
_savegpr0_13 stw r13,-76(r1) #save r13
_savegpr0_14 stw r14,-72(r1) #save r14
_savegpr0_15 stw r15,-68(r1) #save r15
_savegpr0_16 stw r16,-64(r1) #save r16
_savegpr0_17 stw r17,-60(r1) #save r17
_savegpr0_18 stw r18,-56(r1) #save r18
_savegpr0_19 stw r19,-52(r1) #save r19
_savegpr0_20 stw r20,-48(r1) #save r20
_savegpr0_21 stw r21,-44(r1) #save r21
_savegpr0_22 stw r22,-40(r1) #save r22
_savegpr0_23 stw r23,-36(r1) #save r23
_savegpr0_24 stw r24,-32(r1) #save r24
_savegpr0_25 stw r25,-28(r1) #save r25
_savegpr0_26 stw r26,-24(r1) #save r26
_savegpr0_27 stw r27,-20(r1) #save r27
_savegpr0_28 stw r28,-16(r1) #save r28
_savegpr0_29 stw r29,-12(r1) #save r29
stw r30,-8(r1) #save r30
stw r31,-4(r1) #save r31
stw r0, 8(r1) #save LR in #caller's frame
blr #return
```

**Note:** This save routine must not be called when GPR30 or GPR31, or both, are the only registers being saved. In these cases, the saving and restoring must be done inline.

The following example shows a GPR restore routine when FPRs are not saved:
Note: This restore routine must not be called when GPR30 or GPR31, or both, are the only registers being saved. In these cases, the saving and restoring must be done inline.

Saving GPRs and FPRs: For a function that saves and restores \( n \) GPRs and \( m \) FPRs (\( n>2 \) and \( m>2 \)), the saving can be done using individual store and load instructions or by calling system-provided routines as shown in the following example:

```assembly
#The following example shows the prolog/epilog of a function which save n GPRs and m FPRs:
mflr r0   #move LR into GPR 0
subi r12,r1,8*m    #compute GPR save pointer
bla _savegpr1_<32-n> #branch and link to save GPRs
bla _savefpr_<32-m>
stwu r1,<-frame_size>(r1)  #update SP and save caller's SP
...
<save CR if necessary>
...
... #body of function
...
<reload save CR if necessary>
...
<reload caller's SP into r1>  #see note below on
subi r12,r1,8*m    #compute CPR restore pointer
bla _restgpr1_<32-n> #restore GPRs
ba _restfpr_<32-m> #restore FPRs and return
```

Note: The calling function SP can be restored by either adding the frame_size value to the current SP whenever the frame_size is known or by reloading it from offset 0 from the current SP. The first approach is more efficient, but not possible for functions that use the alloc subroutine to dynamically allocate stack space.

The following example shows a GPR save routine when FPRs are saved:

```assembly
_savegpr1_13 stw r13,-76(r12)  #save r13
_savegpr1_14 stw r14,-72(r12)  #save r14
_savegpr1_15 stw r15,-68(r12)  #save r15
_savegpr1_16 stw r16,-64(r12)  #save r16
_savegpr1_17 stw r17,-60(r12)  #save r17
_savegpr1_18 stw r18,-56(r12)  #save r18
_savegpr1_19 stw r19,-52(r12)  #save r19
_savegpr1_20 stw r20,-48(r12)  #save r20
```
The following example shows an FPR save routine:

```
_Printffpr_14 stfd f14,-144(r1) #save f14
_Printffpr_15 stfd f15,-136(r1) #save f15
_Printffpr_16 stfd f16,-128(r1) #save f16
_Printffpr_17 stfd f17,-120(r1) #save f17
_Printffpr_18 stfd f18,-112(r1) #save f18
_Printffpr_19 stfd f19,-104(r1) #save f19
_Printffpr_20 stfd f20,-96(r1) #save f20
_Printffpr_21 stfd f21,-88(r1) #save f21
_Printffpr_22 stfd f22,-80(r1) #save f22
_Printffpr_23 stfd f23,-72(r1) #save f23
_Printffpr_24 stfd f24,-64(r1) #save f24
_Printffpr_25 stfd f25,-56(r1) #save f25
_Printffpr_26 stfd f26,-48(r1) #save f26
_Printffpr_27 stfd f27,-40(r1) #save f27
_Printffpr_28 stfd f28,-32(r1) #save f28
_Printffpr_29 stfd f29,-24(r1) #save f29
    stfd f30,-16(r1) #save f30
    stfd f31,-8(r1) #save f31
    stw r0,8(r1) #save LR in
        #caller's frame
    blr #return
```

The following example shows an GPR save routine when FPRs are saved:

```
_Printfgpr_12 stw r21,-44(r12) #save r21
_Printfgpr_13 stw r22,-40(r12) #save r22
_Printfgpr_14 stw r23,-36(r12) #save r23
_Printfgpr_15 stw r24,-32(r12) #save r24
_Printfgpr_16 stw r25,-28(r12) #save r25
_Printfgpr_17 stw r26,-24(r12) #save r26
_Printfgpr_18 stw r27,-20(r12) #save r27
_Printfgpr_19 stw r28,-16(r12) #save r28
_Printfgpr_20 stw r29,-12(r12) #save r29
_Printfgpr_21 stw r30,-8(r12) #save r30
_Printfgpr_22 stw r31,-4(r12) #save r31
_Printfgpr_23 blr #return
```

The following example shows a GPR restore routine when FPRs are saved:

```
_Printfgpr_13 lwz r13,-76(r12) #restore r13
_Printfr_gpr_14 lwz r14,-72(r12) #restore r14
_Printfr_gpr_15 lwz r15,-68(r12) #restore r15
_Printfr_gpr_16 lwz r16,-64(r12) #restore r16
_Printfr_gpr_17 lwz r17,-60(r12) #restore r17
_Printfr_gpr_18 lwz r18,-56(r12) #restore r18
_Printfr_gpr_19 lwz r19,-52(r12) #restore r19
_Printfr_gpr_20 lwz r20,-48(r12) #restore r20
_Printfr_gpr_21 lwz r21,-44(r12) #restore r21
_Printfr_gpr_22 lwz r22,-40(r12) #restore r22
_Printfr_gpr_23 lwz r23,-36(r12) #restore r23
_Printfr_gpr_24 lwz r24,-32(r12) #restore r24
_Printfr_gpr_25 lwz r25,-28(r12) #restore r25
_Printfr_gpr_26 lwz r26,-24(r12) #restore r26
_Printfr_gpr_27 lwz r27,-20(r12) #restore r27
_Printfr_gpr_28 lwz r28,-16(r12) #restore r28
_Printfr_gpr_29 lwz r29,-12(r12) #restore r29
_Printfr_gpr_30 lwz r30,-8(r12) #restore r30
_Printfr_gpr_31 lwz r31,-4(r12) #restore r31
_Printfr_gpr_32 blr #return
```

The following example shows an FPR restore routine:

```
_Printffpr_14 lfd r14,-144(r1) #restore r14
_Printffpr_15 lfd r15,-136(r1) #restore r15
_Printffpr_16 lfd r16,-128(r1) #restore r16
_Printffpr_17 lfd r17,-120(r1) #restore r17
_Printffpr_18 lfd r18,-112(r1) #restore r18
_Printffpr_19 lfd r19,-104(r1) #restore r19
_Printffpr_20 lfd r20,-96(r1) #restore r20
_Printffpr_21 lfd r21,-88(r1) #restore r21
_Printffpr_22 lfd r22,-80(r1) #restore r22
_Printffpr_23 lfd r23,-72(r1) #restore r23
_Printffpr_24 lfd r24,-64(r1) #restore r24
_Printffpr_25 lfd r25,-56(r1) #restore r25
_Printffpr_26 lfd r26,-48(r1) #restore r26
_Printffpr_27 lfd r27,-40(r1) #restore r27
_Printffpr_28 lfd r28,-32(r1) #restore r28
_Printffpr_29 lfd r29,-24(r1) #restore r29
_Printffpr_30 lfd r30,-16(r1) #restore r30
_Printffpr_31 lfd r31,-8(r1) #restore r31
_Printffpr_32 lfd r32,-4(r1) #restore r32
_Printffpr_33 blr #return
```
Saving FPRs Only: For a function that saves and restores \( m \) FPRs \((m>2)\), the saving can be done using individual store and load instructions or by calling system-provided routines as shown in the following example:

```assembly
#The following example shows the prolog/epilog of a function which saves \( m \) FPRs and no GPRs:
mflr  r0   #move LR into GPR 0
bla    _savefpr_<32-m>
stwu  r1,<-frame_size>(r1)  #update SP and save caller's SP
...
<save CR if necessary>
...
#body of function
...
<reload save CR if necessary>
...
<reload caller's SP into r1>  #see note below
ba    _restfpr_<32-m>  #restore FPRs and return
```

Notes:

1. There are no entry points for saving and restoring GPR and FPR numbers higher than 29. It is more efficient to save a small number of registers in the prolog than to call the save and restore functions.
2. The restoring of the calling function SP can be done by either adding the \( \text{frame}\_\text{size} \) value to the current SP whenever \( \text{frame}\_\text{size} \) is known, or by reloading it from offset 0 from the current SP. The first approach is more efficient, but not possible for functions that use the \text{alloca} subroutine to dynamically allocate stack space.

Updating the Stack Pointer: The PowerPC \text{stwu} (Store Word with Update) instruction is used for computing the new SP and saving the back chain. This instruction has a signed 16-bit displacement field that can represent a maximum signed value of 32,768. A stack frame size greater than 32K bytes requires two instructions to update the SP, and the update must be done atomically.

The two assembly code examples illustrate how to update the SP in a prolog:

To compute a new SP and save the old SP for stack frames larger than or equal to 32K bytes:

```assembly
addis r12, r0, (<-frame_size> > 16) & 0xFFFF
# set r12 to left half of frame size
ori r12, r12 (-frame_size> & 0xFFFF
# Add right halfword of frame size
stwux r1, r1, r12  # save old SP and compute new SP
```

To compute a new SP and save the old SP for stack frames smaller than 32K bytes:

```assembly
stwu r1, <-frame_size>(r1)  #update SP and save caller's SP
```
Calling Routine’s Responsibilities

When an assembler language program calls another program, the caller should not use the names of the called program’s commands, functions, or procedures as global assembler language symbols. To avoid confusion, follow the naming conventions for the language of the called program when you create symbol names. For example, if you are calling a C language program, be certain you use the naming conventions for that language.

A called routine has two symbols associated with it: a function descriptor (Name) and an entry point (.Name). When a call is made to a routine, the compiler branches to the name point directly.

Except for when loading parameters into the proper registers, calls to functions are expanded by compilers to include an NOP instruction after each branch and link instruction. This extra instruction is modified by the linkage editor to restore the contents of the TOC register (register 2) on return from an out-of-module call.

The instruction sequence produced by compilers is:

```
bl .foo    #Branch to foo
cror 31,31,31  #Special NOP 0x4ffffb82
```

**Note:** Some compilers produce a `cror 15,15,15` (0x4def7b82) instruction. To avoid having to restore condition register 15 after a call, the linkage editor transforms `cror 15,15,15` into `cror 31,31,31`. Condition register bit 31 is not preserved across a call and does not have to be restored.

The linkage editor will do one of two things when it sees the `bl` instruction (in the previous instruction sequence, on a call to the `foo` function):

- If the `foo` function is imported (not in the same executable module), the linkage editor:
  - Changes the `bl .foo` instruction to `bl .glink_of_foo` (a global linkage routine).
  - Inserts the `.glink` code sequence into the `/usr/lib/glink.o` module.
  - Replaces the NOP `cror` instruction with an `l` (load) instruction to restore the TOC register.

The `bl .foo` instruction sequence is changed to:

```
bl .glink_of_foo    #Branch to global linkage routine for foo
l 2,20(1)          #Restore TOC register instruction 0x80410014
```

- If the `foo` function is bound in the same executable module as its caller, the linkage editor:
  - Changes the `bl .glink_of_foo` sequence (a global linkage routine) to `bl .foo`.
  - Replaces the restore TOC register instruction with the special NOP `cror` instruction.

The `bl .glink_of_foo` instruction sequence is changed to:

```
bl .foo    #Branch to foo
cror 31,31,31  #Special NOP instruction 0x4ffffb82
```

**Note:** For any export, the linkage editor inserts the procedure’s descriptor into the module.

Called Routine’s Responsibilities

Prologs and epilogos are used in the called routines. On entry to a routine, the following steps should be performed:

1. Use some or all of the prolog actions described in the `Prolog Actions` table.
2. Store the back chain and decrement the stack pointer (SP) by the size of the stack frame.

**Note:** If a stack overflow occurs, it will be known immediately when the store of the back chain is completed.

On exit from a procedure, use some or all of the epilog actions described in the `Epilog Actions` table.
Traceback Tags

Every assembly (compiled) program needs traceback information for the debugger to examine if the program traps or crashes during execution. This information is in a traceback table at the end of the last machine instruction in the program and before the program's constant data.

The traceback table starts with a full word of zeros, X'00000000', which is not a valid system instruction. The zeros are followed by 2 words (64 bits) of mandatory information and several words of optional information, as defined in the /usr/include/sys/debug.h file. Using this traceback information, the debugger can unwind the CALL chain and search forward from the point where the failure occurred until it reaches the end of the program (the word of zeros).

In general, the traceback information includes the name of the source language and information about registers used by the program, such as which general-purpose and floating-point registers were saved.

Example

The following is an example of assembler code called by a C routine:

```c
#include <stdio.h>

extern void examlinkage();

void callfile() {
    examlinkage();
}

int main() {
    callfile();
    printf("Hello, world!");
    return 0;
}
```

Compile as follows:

```
cc -o callfile callfile.c examlinkage.s
```

---

The following routine calls printf() to print a string.

```
extern void examlinkage();

void callfile() {
    printf("Hello, world!");
    return;
}
```

The routine performs entry steps 1-5 and exit steps 1-6.

The prolog/epilog code is for small stack frame size.

```
file "examlinkage.s"
```

---

Assembler Language Reference
#Static data entry in TableOfContents
.toc
T.examlinkage.c: .tc examlinkage.c[tc],examlinkage.c[rw]
.globl examlinkage[ds]
#examlinkage[ds] contains definitions needed for
#runtime linkage of function examlinkage
.csect examlinkage[ds]
.long examlinkage[PR]
.long TOC[tc0]
.long 0
#Function entry in TableOfContents
.toc
T.examlinkage: .tc examlinkage[tc],examlinkage[ds]
#Main routine
.globl examlinkage[PR]
.csect examlinkage[PR]
# Set current routine stack variables
# These values are specific to the current routine and
# can vary from routine to routine
.set argarea, 32
.set linkarea, 24
.set locstckarea, 0
.set nfprs, 18
.set ngprs, 19
.set szdsa, 8*nfprs+4*ngprs+linkarea+argarea+locstckarea
#PROLOG: Called Routines Responsibilities
# Get link reg.
mflr 0
# Get CR if current routine alters it.
mfcr 12
bl _savef14
cror 31, 31, 31
# Save GPRs 13-31.
stm 13, -8*nfprs-4*ngprs(1)
# Save LR if non-leaf routine.
st 0, 8(1)
# Save CR if current routine alters it.
st 12, 4(1)
# Decrement stack ptr and save back chain.
stu 1, -szdsa(1)

Chapter 5. Assembling and Linking a Program
#EPILOG: Return Sequence
# Restore stack ptr
ai 1, 1, szdsa
# Restore GPRs 13-31.
1m 13, -8*nfprs-4*ngprs(1)
# Restore FPRs 14-31.
bl _restf14
cror 31, 31, 31
# Get saved LR.
1 0, 8(1)
# Get saved CR if this routine saved it.
1 12, 4(1)
# Move return address to link register.
mflr 0
# Restore CR2, CR3, & CR4 of the CR.
mtnf 0x38,12
Using Milicode Routines

All of the fixed-point divide instructions, and some of the multiply instructions, are different for POWER family and PowerPC. To allow programs to run on systems based on either architecture, a set of special routines is provided by the operating system. These are called milicode routines and contain machine-dependent and performance-critical functions. Milicode routines are located at fixed addresses in the kernel segment. These routines can be reached by a `bla` instruction. All milicode routines use the link register.

Notes:
1. No unnecessary registers are destroyed. Refer to the definition of each milicode routine for register usage information.
2. Milicode routines do not alter any floating-point register, count register, or general-purpose registers (GPRs) 10-12. The link register can be saved in a GPR (for example, GPR 10) if the call appears in a leaf procedure that does not use nonvolatile GPRs.
3. Milicode routines do not make use of a TOC.

The following milicode routines are available:

__mulh__
Calculates the high-order 32 bits of the integer product \( \text{arg1} \times \text{arg2} \).

Input
- \( R3 = \text{arg1} \) (signed integer)
- \( R4 = \text{arg2} \) (signed integer)

Output
- \( R3 = \) high-order 32 bits of \( \text{arg1} \times \text{arg2} \)

POWER family Register Usage
- GPR3, GPR4, MQ

PowerPC Register Usage
- GPR3, GPR4

__mull__
Calculates 64 bits of the integer product \( \text{arg1} \times \text{arg2} \), returned in two 32-bit registers.

Input
- \( R3 = \text{arg1} \) (signed integer)
- \( R4 = \text{arg2} \) (signed integer)

Output
- \( R3 = \) high-order 32 bits of \( \text{arg1} \times \text{arg2} \)
- \( R4 = \) low-order 32 bits of \( \text{arg1} \times \text{arg2} \)

POWER family Register Usage
- GPR3, GPR4, MQ

PowerPC Register Usage
- GPR0, GPR3, GPR4
__divss  Calculates the 32-bit quotient and 32-bit remainder of signed integers \( \text{arg1}/\text{arg2} \). For division by zero and overflow, the quotient and remainder are undefined and may vary by implementation.

**Input**  
\[ R3 = \text{arg1} \text{ (dividend) (signed integer)} \]  
\[ R4 = \text{arg2} \text{ (divisor) (signed integer)} \]

**Output**  
\[ R3 = \text{quotient of arg1/arg2 (signed integer)} \]  
\[ R4 = \text{remainder of arg1/arg2 (signed integer)} \]

**POWER family Register Usage**  
GPR3, GPR4, MQ

**PowerPC Register Usage**  
GPR0, GPR3, GPR4

__divus  Calculated the 32-bit quotient and 32-bit remainder of unsigned integers \( \text{arg1}/\text{arg2} \). For division by zero and overflow, the quotient and remainder are undefined and may vary by implementation.

**Input**  
\[ R3 = \text{arg1} \text{ (dividend) (unsigned integer)} \]  
\[ R4 = \text{arg2} \text{ (divisor) (unsigned integer)} \]

**Output**  
\[ R3 = \text{quotient of arg1/arg2 (unsigned integer)} \]  
\[ R4 = \text{remainder of arg1/arg2 (unsigned integer)} \]

**POWER family Register Usage**  
GPR0, GPR3, GPR4, MQ, CR0 and CR1 of CR

**PowerPC Register Usage**  
GPR0, GPR3, GPR4

__quoss  Calculates the 32-bit quotient of signed integers \( \text{arg1}/\text{arg2} \). For division by zero and overflow, the quotient and remainder are undefined and may vary by implementation.

**Input**  
\[ R3 = \text{arg1} \text{ (dividend) (signed integer)} \]  
\[ R4 = \text{arg2} \text{ (divisor) (signed integer)} \]

**Output**  
\[ R3 = \text{quotient of arg1/arg2 (signed integer)} \]

**POWER family Register Usage**  
GPR3, GPR4, MQ

**PowerPC Register Usage**  
GPR3, GPR4

__quous  Calculates the 32-bit quotient of unsigned integers \( \text{arg1}/\text{arg2} \). For division by zero and overflow, the quotient and remainder are undefined and may vary by implementation.

**Input**  
\[ R3 = \text{arg1} \text{ (dividend) (unsigned integer)} \]  
\[ R4 = \text{arg2} \text{ (divisor) (unsigned integer)} \]

**Output**  
\[ R3 = \text{quotient of arg1/arg2 (unsigned integer)} \]

**POWER family Register Usage**  
GPR0, GPR3, GPR4, MQ, CR0 and CR1 of CR

**PowerPC Register Usage**  
GPR3, GPR4

The following example uses the \texttt{mulh} milicode routine in an assembler program:

```
li R3, -900
li R4, 50000
bla .__mulh
...
.extern .__mulh
```
Related Information

“Assembling and Linking a Program” on page 53.

“Understanding Assembler Passes” on page 57.

“Interpreting an Assembler Listing” on page 59.

“Interpreting a Symbol Cross-Reference” on page 63.

“Understanding and Programming the TOC.”

“Running a Program” on page 87.

“b (Branch) Instruction” on page 143, “cror (Condition Register OR) Instruction” on page 168.

Understanding and Programming the TOC

The Table of Contents (TOC) of an XCOFF file is analogous to the table of contents of a book. The TOC is used to find objects in an XCOFF file. An XCOFF file is composed of sections that contain different types of data to be used for specific purposes. Some sections can be further subdivided into subsections or csects. A csect is the smallest replaceable unit of an XCOFF file. At run time, the TOC can contain the csect locations (and the locations of labels inside of csects).

The three sections that contain csects are:

- .text: Indicates that this csect contains code or read-only data.
- .data: Indicates that this csect contains read-write data.
- .bss: Indicates that this csect contains uninitialized mapped data.

The storage class of the csect determines the section in which the csect is grouped.

The TOC is located in the .data section of an XCOFF object file and is composed of TOC entries. Each TOC entry is a csect with storage mapping class of TC or TD.

A TOC entry with TD storage mapping class contains scalar data which can be directly accessed from the TOC. This permits some frequently used global symbols to be accessed directly from the TOC rather than indirectly through an address pointer csect contained within the TOC. To access scalar data in the TOC, two pieces of information are required:

- The location of the beginning of the TOC (i.e. the TOC anchor).
- The offset from the TOC anchor to the specific TOC entry that contains the data.

A TOC entry with TC storage mapping class contains the addresses of other csects or global symbols. Each entry can contain one or more addresses of csects or global symbols, but putting only one address in each TOC entry is recommended.

When a program is assembled, the csects are sorted such that the .text csects are written first, followed by all .data csects except for the TOC. The TOC is written after all the other .data csects. The TOC entries are relocated, so that the TOC entries with TC storage mapping class contain the csect addresses after the sort, rather than the csect addresses in the source program.

When an XCOFF module is loaded, TOC entries with TC storage mapping class are relocated again so that the TOC entries are filled with the real addresses where the csects will reside in memory. To access a csect in the module, two pieces of information are required:

- The location of the beginning of the TOC.
• The offset from the beginning of the TOC to the specific TOC entry that points to the csect. If a TOC entry has more than one address, each address can be calculated by adding \((0...(n-1))\times 4\) to the offset, where \(n\) is the position of the csect address defined with the \(\texttt{.tc}\) Pseudo-op on page 503.

### Using the TOC

To use the TOC, you must follow certain conventions:

- General-Purpose Register 2 always contains a pointer to the TOC.
- All references from the \(\texttt{.text}\) section of an assembler program to \(\texttt{.data}\) or the \(\texttt{.bss}\) sections must occur via the TOC.

The TOC register (General-Purpose Register 2) is set up by the system when a program is invoked. It must be maintained by any code written. The TOC register provides module context so that any routines in the module can access data items.

The second of these conventions allows the \(\texttt{.text}\) and \(\texttt{.data}\) sections to be easily loaded into different locations in memory. By following this convention, you can assure that the only parts of the module to need relocating are the TOC entries.

### Accessing Data through the TOC Entry with TC Storage Mapping Class

An external data item is accessed by first getting that item’s address out of the TOC, and then using that address to get the data. In order to do this, proper relocation information must be provided to access the correct TOC entry. The \(\texttt{.toc}\) and \(\texttt{.tc}\) pseudo-ops generate the correct information to access a TOC entry. The following code shows how to access item \(a\) using its TOC entry:

```
.set RTOC,2
.csect prog1[pr]   #prog1 is a csect
                  #containing instrs.
...                
1 5,TCA(RTOC)      #Now GPR5 contains the #address of a[rw].
...

TCA: .tc a[tc],a[rw] #1st parameter is TOC entry
       #name, 2nd is contents of
       #TOC entry.
       .extern a[rw]    #a[rw] is an external symbol.
```

This same method is used to access a program’s static internal data, which is data that retains its value over a call, but which can only be accessed by the procedures in the file where the data items are declared. Following is the C language data having the \texttt{static} attribute:

```
static int xyz;
```

This data is given a name determined by convention. In XCOFF, the name is preceded by an underscore:

```
.csect prog1[pr]
...                
1 1,STprog1(RTOC)  #Load r1 with the address
                  #prog1's static data.
...                
.csect _prog1[wr]  #prog1's static data.
   .long 0
...                
STprog1: .tc.prog1[tc],_prog1[wr]  #TOC entry with address of
                                  #prog1's static data.
```

### Accessing Data through the TOC entry with TD Storage Mapping Class

A scalar data item can be stored into a TOC entry with TD storage mapping class and retrieved directly from the TOC entry.
Note: TOC entries with TD storage mapping class should be used only for frequently used scalars. If the TOC grows too big (either because of many entries or because of large entries) the assembler may report message 1252-171 indicating an out of range displacement.

The following examples show several ways to store and retrieve a scalar data item as a TOC with TD storage mapping class. Each example includes C source for a main program, assembler source for one module, instructions for linking and assembling, and output from running the program.

Example Using .csect Pseudo-op with TD Storage Mapping Class

1. The following is the source for the C main program td1.c:

```c
/* This C module named td1.c */
extern long t_data;
extern void mod_s();
main()
{
    mod_s();
    printf("t_data is %d\n", t_data);
}
```

2. The following is the assembler source for module mod1.s:

```assembly
.file "mod1.s"
csect .mod_s[PR]
globl .mod_s[PR]
.set RTOC, 2
l 5, t_data[TD](RTOC) # Now GPR5 contains the
# t_data value 0x10
ai 5,5,14
stu 5, t_data[TD](RTOC)
br
globl t_data[TD]
toc
csect t_data[TD] # t_data is a global symbol
# that has value of 0x10
# using TD csect will put this
# data into TOC area
.long 0x10
```

3. The following commands assemble and compile the source programs into an executable td1:

`as -o mod1.o mod1.s
cc -o td1 td1.c mod1.o`

4. Running td1 prints the following:

`t_data is 30`

Example Using .comm Pseudo-op with TD Storage Mapping Class

1. The following is the source for the C main program td2.c:

```c
/* This C module named td2.c */
extern long t_data;
extern void mod_s();
main()
{
    t_data = 1234;
    mod_s();
    printf("t_data is %d\n", t_data);
}
```

2. The following is the assembler source for module mod2.s:

```assembly
.file "mod2.s"
csect .mod_s[PR]
globl .mod_s[PR]
.set RTOC, 2
l 5, t_data[TD](RTOC) # Now GPR5 contains the
# t_data value
ai 5,5,14
```
3. The following commands assemble and compile the source programs into an executable td2:
   as -o mod2.o mod2.s  
   cc -o td2 td2.c mod2.o

4. Running td2 prints the following:
   t_data is 1248

**Example Using an External TD Symbol**

1. /* This C module named td3.c */
   long t_data;
   extern void mod_s();
   main()
   {
      t_data = 234;
      mod_s();
      printf("t_data is %d\n", t_data);
   }

2. The following is the assembler source for module mod3.s:
   .file "mod3.s"
   .csect .mod_s[PR]
   .globl .mod_s[PR]
   .set  RTOC, 2
   l 5, t_data[TD](RTOC)  # Now GPR5 contains the
   # t_data value
   ai 5,5,14
   stu 5, t_data[TD](RTOC)
   br 
   .toc
   .extern t_data[TD]  # t_data is a external symbol

3. The following commands assemble and compile the source programs into an executable td3:
   ./as -o mod3.o mod3.s  
   cc -o td3 td3.c mod3.o

4. Running td3 prints the following:
   t_data is 248

**Intermodule Calls Using the TOC**

Because the only access from the text to the data section is through the TOC, the TOC provides a feature that allows intermodule calls to be used. As a result, routines can be linked together without resolving all the addresses or symbols at link time. In other words, a call can be made to a common utility routine without actually having that routine linked into the same module as the calling routine. In this way, groups of routines can be made into modules, and the routines in the different groups can call each other, with the bind time being delayed until load time. In order to use this feature, certain conventions must be followed when calling a routine that is in another module.

To call a routine in another module, an interface routine (or **global linkage** routine) is called that switches context from the current module to the new module. This context switch is easily performed by saving the TOC pointer to the current module, loading the TOC pointer of the new module, and then branching to the new routine in the other module. The other routine then returns to the original routine in the original module, and the original TOC address is loaded into the TOC register.

To make global linkage as transparent as possible, a call can be made to external routines without specifying the destination module. During bind time, the binder (linkage editor) determines whether to call global linkage code, and inserts the proper global linkage routine to perform the intermodule call. Global
linkage is controlled by an import list. An import list contains external symbols that are resolved during runtime, either from the system or from the dynamic load of another object file. See the \texttt{id} command for information about import lists.

The following example calls a routine that may go through global linkage:

\begin{verbatim}
.csect prog1[PR] ...
.extern prog2[PR]        #prog2 is an external symbol.
bl  .prog2[PR]          #call prog2[PR], binder may insert
                        #global linkage code.
cror 31,31,31          #place holder for instruction to
                        #restore TOC address.
\end{verbatim}

The following example shows a call through a global linkage routine:

\begin{verbatim}
#AIX linkage register conventions:
# R2  TOC
# R1  stack pointer
# R0, R12 work registers, not preserved
# LR  Link Register, return address.
.csect .prog1[PR]
.bl .prog2[GL]        #Branch to global
                        #linkage code.
.l 2,stktoc(1)       #Restore TOC address
.toc
_prog2: .tc prog2[TC],prog2[DS] #TOC entry:
                        #address of descriptor
                        #for out-of-module
                        #routine
.extern prog2[DS]
##
## The following is an example of global linkage code.
.set stktoc,20
.csect .prog2[GL]
.globl .prog2
_prog2: l 12,prog2(2)   #Get address of
                        #out-of-module
                        #descriptor.
.st 2,stktoc(1)        #Save callers' toc.
.l 0,0(12)             #Get its entry address
                        #from descriptor.
.l 2,4(12)             #Get its toc from
                        #descriptor.
.mtcrt 0               #Put into Count Register.
.btcr                   #Return to entry address
                        #in Count Register.
                        #Return is directly to
                        #original caller.
\end{verbatim}

Related Information

"Assembling and Linking a Program" on page 53.

"Understanding Assembler Passes" on page 57.

"Interpreting an Assembler Listing" on page 59.

"Interpreting a Symbol Cross-Reference" on page 63.

"Subroutine Linkage Convention" on page 65.

"Running a Program" on page 87.
Running a Program

A program is ready to run when it has been assembled and linked without producing any error messages. To run a program, first ensure that you have operating system permission to execute the file. Then type the program's name at the operating system prompt:

```
$ progname
```

By default, any program output goes to standard output. To direct output somewhere other than standard output, use the operating system shell `>` (more than symbol) operator.

Run-time errors can be diagnosed by invoking the symbolic debugger with the `dbx` command. This symbolic debugger works with any code that adheres to XCOFF format conventions. The `dbx` command can be used to debug all compiler- and assembler-generated code.

Related Information

- “Assembling and Linking a Program” on page 53.
- “Understanding Assembler Passes” on page 57.
- “Interpreting an Assembler Listing” on page 59.
- “Interpreting a Symbol Cross-Reference” on page 63.
- “Subroutine Linkage Convention” on page 65.
- “Understanding and Programming the TOC” on page 82.

The `dbx` command.

The `as` command, `dbx` command, `ld` command.

- “b (Branch) Instruction” on page 143.
- “cror (Condition Register OR) Instruction” on page 168.
Chapter 6. Extended Instruction Mnemonics

The assembler supports a set of extended mnemonics and symbols to simplify assembly language programming. All extended mnemonics should be in the same assembly mode as their base mnemonics. Although different extended mnemonics are provided for POWER family and PowerPC, the assembler generates the same object code for the extended mnemonics if the base mnemonics are in the com assembly mode. The assembly mode for the extended mnemonics are listed in each extended mnemonics section. The POWER family and PowerPC extended mnemonics are listed separately in the following sections for migration purposes:

- "Extended Mnemonics of Branch Instructions"
- "Extended Mnemonics of Condition Register Logical Instructions" on page 96
- "Extended Mnemonics of Fixed-Point Arithmetic Instructions" on page 97
- "Extended Mnemonics of Fixed-Point Compare Instructions" on page 98
- "Extended Mnemonics of Fixed-Point Load Instructions" on page 99
- "Extended Mnemonics of Fixed-Point Logical Instructions" on page 100
- "Extended Mnemonics of Fixed-Point Trap Instructions" on page 100
- "Extended Mnemonic mtcr for Moving to the Condition Register" on page 102
- "Extended Mnemonics of Moving from or to Special-Purpose Registers" on page 102
- "Extended Mnemonics of 32-bit Fixed-Point Rotate and Shift Instructions" on page 107

Extended Mnemonics of Branch Instructions

The assembler supports extended mnemonics for Branch Conditional, Branch Conditional to Link Register, and Branch Conditional to Count Register instructions. Since the base mnemonics for all the Branch Conditional instructions are in the com assembly mode, all of their extended mnemonics are also in the com assembly mode.

Extended mnemonics are constructed by incorporating the BO and BI input operand into the mnemonics. Extended mnemonics always omit the BH input operand and assume its value to be 0b00.

Branch Mnemonics That Incorporate Only the BO Operand

The following tables show the instruction format for extended mnemonics that incorporate only the BO field. The target address is specified by the target_addr operand. The bit in the condition register for condition comparison is specified by the BI operand. The value of the BI operand can be specified by an expression. The CR field number should be multiplied by four to get the correct CR bit, since each CR field has four bits.

Note: Some extended mnemonics have two input operand formats.

Table 7. POWER family Extended Mnemonics (BO Field Only)

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Input Operands</th>
<th>Equivalent to</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bdz, bdza, bdzl, bdzia</td>
<td>target_addr</td>
<td>bc, bca, bcl, bcla</td>
<td>18, 0, target_addr</td>
</tr>
<tr>
<td>bdn, bdna, bdnl, bdnla</td>
<td>target_addr</td>
<td>bc, bca, bcl, bcla</td>
<td>16, 0, target_addr</td>
</tr>
<tr>
<td>bdzr, bdzrl</td>
<td>None</td>
<td>bcr, bcrl</td>
<td>18, 0</td>
</tr>
<tr>
<td>bdnr, bdnrl</td>
<td>None</td>
<td>bcr, bcrl</td>
<td>16, 0</td>
</tr>
<tr>
<td>bbt, bbta, bbtl, bbtlra</td>
<td>1) BI, target_addr</td>
<td>bc, bca, bcl, bcla</td>
<td>12, BI, target_addr</td>
</tr>
<tr>
<td></td>
<td>2) target_addr</td>
<td></td>
<td>12, 0, target_addr</td>
</tr>
<tr>
<td>bbf, bbfa, bbfl, bbfla</td>
<td>1) BI, target_addr</td>
<td>bc, bca, bcl, bcla</td>
<td>4, BI, target_addr</td>
</tr>
<tr>
<td></td>
<td>2) target_addr</td>
<td></td>
<td>4, 0, target_addr</td>
</tr>
</tbody>
</table>
Table 7. POWER family Extended Mnemonics (BO Field Only) (continued)

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Input Operands</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbtr, bbtc, bbtrl, bbctl</td>
<td>1) BL</td>
<td>bcr, bcc, bcrl, bccl 12, BL</td>
</tr>
<tr>
<td></td>
<td>2) None</td>
<td>12, 0</td>
</tr>
<tr>
<td>bbfr, bbfc, bbfrl, bbfcl</td>
<td>1) BL</td>
<td>bcr, bcc, bcrl, bccl 4, BL</td>
</tr>
<tr>
<td></td>
<td>2) None</td>
<td>4, 0</td>
</tr>
<tr>
<td>br, bctr, brl, bctl</td>
<td>None</td>
<td>bcr, bcc, bcrl, bccl 20, 0</td>
</tr>
</tbody>
</table>

Table 8. PowerPC Extended Mnemonics (BO Field Only)

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Input Operands</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>bdz, bdza, bdzl, bdzla</td>
<td>target_addr</td>
<td>bc, bca, bcl, bcla 18, 0, target_addr</td>
</tr>
<tr>
<td>bdnz, bdnza, bdnzl, bdnzla</td>
<td>target_addr</td>
<td>bc, bca, bcl, bcla 16, 0, target_addr</td>
</tr>
<tr>
<td>bdzlr, bdzlrl</td>
<td>None</td>
<td>bclr, bcrlr 18, 0</td>
</tr>
<tr>
<td>bdnzlr, bdnzlrl</td>
<td>None</td>
<td>bclr, bcrlr 16, 0</td>
</tr>
<tr>
<td>bt, bta, btl, btlr</td>
<td>1) BL, target_addr</td>
<td>bc, bca, bcl, bcla 12, BI, target_addr</td>
</tr>
<tr>
<td></td>
<td>2) target_addr</td>
<td>12, 0, target_addr</td>
</tr>
<tr>
<td>bf, bfa, bfl, bfra</td>
<td>1) BL, target_addr</td>
<td>bc, bca, bcl, bcla 4, BI, target_addr</td>
</tr>
<tr>
<td></td>
<td>2) target_addr</td>
<td>4, 0, target_addr</td>
</tr>
<tr>
<td>bdzt, bdzta, bdztl, bdztlr</td>
<td>1) BL, target_addr</td>
<td>bc, bca, bcl, bcla 10, BI, target_addr</td>
</tr>
<tr>
<td></td>
<td>2) target_addr</td>
<td>10, 0, target_addr</td>
</tr>
<tr>
<td>bdzf, bdzfna, bdzfli, bdzfli</td>
<td>1) BL, target_addr</td>
<td>bc, bca, bcl, bcla 2, BI, target_addr</td>
</tr>
<tr>
<td></td>
<td>2) target_addr</td>
<td>2, 0, target_addr</td>
</tr>
<tr>
<td>bdnzt, bdnzta, bdnztl, bdnztlr</td>
<td>1) BL, target_addr</td>
<td>bc, bca, bcl, bcla 8, BI, target_addr</td>
</tr>
<tr>
<td></td>
<td>2) target_addr</td>
<td>8, 0, target_addr</td>
</tr>
<tr>
<td>bdnzf, bdnzfa, bdnzfl, bdnzflr</td>
<td>1) BL, target_addr</td>
<td>bc, bca, bcl, bcla 0, BI, target_addr</td>
</tr>
<tr>
<td></td>
<td>2) target_addr</td>
<td>0, 0, target_addr</td>
</tr>
<tr>
<td>btlr, btlcr, btlr, bcctrlr</td>
<td>1) BL</td>
<td>bclr, bcctr, bcrlr, bcctrlr 12, BI</td>
</tr>
<tr>
<td></td>
<td>2) None</td>
<td>12, 0</td>
</tr>
<tr>
<td>bflr, bflcr, bflr, bcflcr</td>
<td>1) BL</td>
<td>bclr, bcctr, bcrlr, bcctrl 4, BI</td>
</tr>
<tr>
<td></td>
<td>2) None</td>
<td>4, 0</td>
</tr>
<tr>
<td>bdztlr, bdztlrl</td>
<td>1) BL</td>
<td>bclr, bcrlr 10, BI</td>
</tr>
<tr>
<td></td>
<td>2) None</td>
<td>10, 0</td>
</tr>
<tr>
<td>bdzfir, bdzfirl</td>
<td>1) BL</td>
<td>bclr, bcrlr 2, BI</td>
</tr>
<tr>
<td></td>
<td>2) None</td>
<td>2, 0</td>
</tr>
<tr>
<td>bdnztlr, bdnztlrl</td>
<td>1) BL</td>
<td>bclr, bcrlr 8, BI</td>
</tr>
<tr>
<td></td>
<td>2) None</td>
<td>8, 0</td>
</tr>
<tr>
<td>bdnzflr, bdnzflrl</td>
<td>1) BL</td>
<td>bclr, bcrlr 0, BI</td>
</tr>
<tr>
<td></td>
<td>2) None</td>
<td>0, 0</td>
</tr>
<tr>
<td>blr, bctr, blr, bctl</td>
<td>None</td>
<td>bclr, bcctr, bcrlr, bcctrl 20, 0</td>
</tr>
</tbody>
</table>
Extended Branch Mnemonics That Incorporate the BO Field and a Partial BI Field

When the BO field and a partial BI field are incorporated, the instruction format is one of the following:
- mnemonic BIF, target_addr
- mnemonic target_addr

where the BIF operand specifies the CR field number (0-7) and the target_addr operand specifies the target address. If CR0 is used, the BIF operand can be omitted.

Based on the bits definition in the CR field, the following set of codes has been defined for the most common combinations of branch conditions:

<table>
<thead>
<tr>
<th>Branch Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt</td>
<td>less than *</td>
</tr>
<tr>
<td>eq</td>
<td>equal to *</td>
</tr>
<tr>
<td>gt</td>
<td>greater than *</td>
</tr>
<tr>
<td>so</td>
<td>summary overflow *</td>
</tr>
<tr>
<td>le</td>
<td>less than or equal to * (not greater than)</td>
</tr>
<tr>
<td>ge</td>
<td>greater than or equal to * (not less than)</td>
</tr>
<tr>
<td>ne</td>
<td>not equal to *</td>
</tr>
<tr>
<td>ns</td>
<td>not summary overflow *</td>
</tr>
<tr>
<td>nl</td>
<td>not less than</td>
</tr>
<tr>
<td>ng</td>
<td>not greater than</td>
</tr>
<tr>
<td>z</td>
<td>zero</td>
</tr>
<tr>
<td>nu</td>
<td>not unordered (after floating-point comparison)</td>
</tr>
<tr>
<td>nz</td>
<td>not zero</td>
</tr>
<tr>
<td>un</td>
<td>unordered (after floating-point comparison)</td>
</tr>
</tbody>
</table>

The assembler supports six encoding values for the BO operand:
- Branch if condition true (BO=12):

<table>
<thead>
<tr>
<th>POWER family</th>
<th>PowerPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>bxx</td>
<td>bxx</td>
</tr>
<tr>
<td>bxxa</td>
<td>bxxa</td>
</tr>
<tr>
<td>bxxl</td>
<td>bxxl</td>
</tr>
<tr>
<td>bxxla</td>
<td>bxxla</td>
</tr>
<tr>
<td>bxxr</td>
<td>bxxr</td>
</tr>
<tr>
<td>bxxrl</td>
<td>bxxrl</td>
</tr>
<tr>
<td>bxxc</td>
<td>bxxctr</td>
</tr>
<tr>
<td>bxxcl</td>
<td>bxxctr</td>
</tr>
</tbody>
</table>

where xx specifies a BI operand branch code of lt, gt, eq, so, z, or un.

- Branch if condition false (BO=04):

<table>
<thead>
<tr>
<th>POWER family</th>
<th>PowerPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>bxx</td>
<td>bxx</td>
</tr>
<tr>
<td>bxxa</td>
<td>bxxa</td>
</tr>
<tr>
<td>bxxl</td>
<td>bxxl</td>
</tr>
<tr>
<td>bxxla</td>
<td>bxxla</td>
</tr>
<tr>
<td>bxxr</td>
<td>bxxr</td>
</tr>
<tr>
<td>bxxrl</td>
<td>bxxrl</td>
</tr>
<tr>
<td>bxxc</td>
<td>bxxctr</td>
</tr>
<tr>
<td>bxxcl</td>
<td>bxxctr</td>
</tr>
</tbody>
</table>

where xx specifies a BI operand branch code of ge, le, ne, ns, nl, ng, nz, or nu.
• Decrement CTR, then branch if CTR is nonzero and condition is true (\(BO=08\)):
  – \(b\)\(dn\)\(xx\)
  where \(xx\) specifies a BI operand branch code of \(lt\), \(gt\), \(eq\), or \(so\) (marked by an * (asterisk) in the Branch Code list).
• Decrement CTR, then branch if CTR is nonzero and condition is false (\(BO=00\)):
  – \(b\)\(dn\)\(xx\)
  where \(xx\) specifies a BI operand branch code of \(le\), \(ge\), \(ne\), or \(ns\) (marked by an * (asterisk) in the Branch Code list).
• Decrement CTR, then branch if CTR is zero and condition is true (\(BO=10\)):
  – \(b\)\(dz\)\(xx\)
  where \(xx\) specifies a BI operand branch code of \(lt\), \(gt\), \(eq\), or \(so\) (marked by an * (asterisk) in the Branch Code list).
• Decrement CTR, then branch if CTR is zero and condition is false (\(BO=02\)):
  – \(b\)\(dz\)\(xx\)
  where \(xx\) specifies a BI operand branch code of \(le\), \(ge\), \(ne\), or \(ns\) (marked by an * (asterisk) in the Branch Code list).

**BI Operand of Branch Conditional Instructions for Basic and Extended Mnemonics**

The BI operand specifies a bit (0:31) in the Condition Register for condition comparison. The bit is set by a compare instruction. The bits in the Condition Register are grouped into eight 4-bit fields. These fields are named CR field 0 through CR field 7 (CR0...CR7). The bits of each field are interpreted as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Less than; floating-point less than</td>
</tr>
<tr>
<td>1</td>
<td>Greater than; floating-point greater than</td>
</tr>
<tr>
<td>2</td>
<td>Equal; floating-point equal</td>
</tr>
<tr>
<td>3</td>
<td>Summary overflow; floating-point unordered</td>
</tr>
</tbody>
</table>

Normally the symbols shown in the BI Operand Symbols for Basic and Extended Branch Conditional Mnemonics table are defined for use in BI operands. The assembler supports expressions for the BI operands. The expression is a combination of values and the following symbols.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt</td>
<td>0</td>
<td>less than</td>
</tr>
<tr>
<td>gt</td>
<td>1</td>
<td>greater than</td>
</tr>
<tr>
<td>eq</td>
<td>2</td>
<td>equal</td>
</tr>
<tr>
<td>so</td>
<td>3</td>
<td>summary overflow</td>
</tr>
<tr>
<td>un</td>
<td>3</td>
<td>unordered (after floating-point comparison)</td>
</tr>
<tr>
<td>cr0</td>
<td>0</td>
<td>CR field 0</td>
</tr>
<tr>
<td>cr1</td>
<td>1</td>
<td>CR field 1</td>
</tr>
<tr>
<td>cr2</td>
<td>2</td>
<td>CR field 2</td>
</tr>
<tr>
<td>cr3</td>
<td>3</td>
<td>CR field 3</td>
</tr>
<tr>
<td>cr4</td>
<td>4</td>
<td>CR field 4</td>
</tr>
<tr>
<td>cr5</td>
<td>5</td>
<td>CR field 5</td>
</tr>
</tbody>
</table>
Table 9. BI Operand Symbols for Basic and Extended Branch Conditional Mnemonics (continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>cr6</td>
<td>6</td>
<td>CR field 6</td>
</tr>
<tr>
<td>cr7</td>
<td>7</td>
<td>CR field 7</td>
</tr>
</tbody>
</table>

When using an expression for the BI field in the basic or extended mnemonics with only the BO field incorporated, the CR field number should be multiplied by 4 to get the correct CR bit, since each CR field has four bits.

1. To decrement CTR, then branch only if CTR is not zero and condition in CR5 is equal:
   \[ \text{bdnzt} \ 4 \times \text{cr5} \times \text{eq}, \text{target_addr} \]
   This is equivalent to:
   \[ \text{bc} \ 8, 22, \text{target_addr} \]

2. To decrement CTR, then branch only if CTR is not zero and condition in CR0 is equal:
   \[ \text{bdnzt} \ \text{eq}, \text{target_addr} \]
   This is equivalent to:
   \[ \text{bc} \ 8, 2, \text{target_addr} \]

   If the BI operand specifies Bit 0 of CR0, the BI operand can be omitted.

3. To decrement CTR, then branch only if CTR is zero:
   \[ \text{bdz} \ \text{target_addr} \]
   This is equivalent to:
   \[ \text{bc} \ 18, 0, \text{target_addr} \]

For extended mnemonics with the BO field and a partial BI field incorporated, the value of the BI operand indicates the CR field number. Valid values are 0-7. If a value of 0 is used, the BI operand can be omitted.

1. To branch if CR0 reflects a condition of not less than:
   \[ \text{bge} \ \text{target_addr} \]
   This is equivalent to:
   \[ \text{bc} \ 4, 0, \text{target_addr} \]

2. To branch to an absolute target if CR4 indicates greater than, and set the Link register:
   \[ \text{bgtla} \ \text{cr4}, \text{target_addr} \]
   This is equivalent to:
   \[ \text{bcla} \ 12, 17, \text{target_addr} \]

   The BI operand CR4 is internally expanded to 16 by the assembler. After the gt (greater than) is incorporated, the result of the BI field is 17.

**Extended Mnemonics for Branch Prediction**

If the likely outcome (branch or fall through) of a given Branch Conditional instruction is known, the programmer can include this information in the assembler source program by adding a branch prediction suffix to the mnemonic of the instruction. The assembler uses the branch prediction information to determine the value of a bit in the machine instruction. Using a branch prediction suffix may improve the average performance of a Branch Conditional instruction.
The following suffixes can be added to any Branch Conditional mnemonic, either basic or extended:

+  Predict branch to be taken
-  Predict branch not to be taken (fall through)

The branch prediction suffix should be placed immediately after the rest of the mnemonic (with no separator character). A separator character (space or tab) should be used between the branch prediction suffix and the operands.

If no branch prediction suffix is included in the mnemonic, the assembler uses the following default assumptions in constructing the machine instruction:

- For relative or absolute branches (bc[l][a]) with negative displacement fields, the branch is predicted to be taken.
- For relative or absolute branches (bc[l][a]) with nonnegative displacement fields, the branch is predicted not to be taken (fall through predicted).
- For branches to an address in the LR or CTR (bclr[l]) or (bcctr[l]), the branch is predicted not to be taken (fall through predicted).

The portion of the machine instruction which is controlled by the branch prediction suffix is the y bit of the BO field. The y bit is set as follows:

- Specifying no branch prediction suffix, or using the suffix which is the same as the default assumption causes the y bit to be set to 0.
- Specifying a branch prediction suffix which is the opposite of the default assumption causes the y bit to be set to 1.

The following examples illustrate use of branch prediction suffixes:
1. Branch if CR0 reflects condition less than. Executing the instruction will usually result in branching.
   ```
   blt+ target
   ```
2. Branch if CR0 reflects condition less than. Target address is in the Link Register. Executing the instruction will usually result in falling through to the next instruction.
   ```
   bltlr-
   ```

The following is a list of the Branch Prediction instructions that are supported by the AIX assembler:

- `bc+`  `bc-`  `bca+`  `bca-`
- `bcctr+`  `bcctr-`  `bcctr1+`  `bcctr1-`
- `bcl+`  `bcl-`  `bcl+a+`  `bcl-a-`
- `bclr+`  `bclr-`  `bclrl+`  `bclrl-
- `bdneq+`  `bdneq-`  `bdnge+`  `bdnge-
- `bdngt+`  `bdngt-`  `bdnle+`  `bdnle-
- `bdnlr+`  `bdnlr-`  `bdnne+`  `bdnne-
- `bdnns+`  `bdnns-`  `bdnso+`  `bdnso-
- `bdnz+`  `bdnz-`  `bdnza+`  `bdnza-
- `bdnzf+`  `bdnzf-`  `bdnza+`  `bdnza-
- `bdnzfl+`  `bdnzfl-`  `bdnza+`  `bdnza-
- `bdnzflr+`  `bdnzflr-`  `bdnza+`  `bdnza-
- `bdnzr+`  `bdnzr-`  `bdnza+`  `bdnza-
- `bdnzt+`  `bdnzt-`  `bdnza+`  `bdnza-
- `bdnztl+`  `bdnztl-`  `bdnza+`  `bdnza-
- `bdnztlr+`  `bdnztlr-`  `bdnza+`  `bdnza-
- `bdzl+`  `bdzl-`  `bdz+`  `bdz-
- `bdzeq+`  `bdzeq-`  `bdz+`  `bdz-
- `bdzfa+`  `bdzfa-`  `bdz+`  `bdz-
- `bdzfla+`  `bdzfla-`  `bdz+`  `bdz-
- `bdzflr+`  `bdzflr-`  `bdz+`  `bdz-
- `bdzgl+`  `bdzgl-`  `bdz+`  `bdz-
- `bdzla+`  `bdzla-`  `bdz+`  `bdz-
<table>
<thead>
<tr>
<th>mnemonic</th>
<th>mnemonic</th>
<th>mnemonic</th>
<th>mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>bdzlr+</td>
<td>bdzlr-</td>
<td>bdzlr+</td>
<td>bdzlr-</td>
</tr>
<tr>
<td>bdzt+</td>
<td>bdzl-</td>
<td>bdzta+</td>
<td>bdzt-</td>
</tr>
<tr>
<td>bdzt+</td>
<td>bdzt+</td>
<td>bdzta-</td>
<td>bdzt-</td>
</tr>
<tr>
<td>bdztlr+</td>
<td>bdztlr-</td>
<td>bdztlr+</td>
<td>bdztlr-</td>
</tr>
<tr>
<td>beq-</td>
<td>beq+</td>
<td>beq-</td>
<td>beq+</td>
</tr>
<tr>
<td>beqctr+</td>
<td>beqctrl+</td>
<td>beqctr-</td>
<td>beqctrl-</td>
</tr>
<tr>
<td>beql+</td>
<td>beql-</td>
<td>beql+</td>
<td>beql-</td>
</tr>
<tr>
<td>beqlr+</td>
<td>beqlr-</td>
<td>beqlr+</td>
<td>beqlr-</td>
</tr>
<tr>
<td>bf-</td>
<td>bfa-</td>
<td>bfa+</td>
<td>bf-</td>
</tr>
<tr>
<td>bfc+</td>
<td>bfc-</td>
<td>bfc+</td>
<td>bfc-</td>
</tr>
<tr>
<td>bfl+</td>
<td>bfl-</td>
<td>bfl+</td>
<td>bfl-</td>
</tr>
<tr>
<td>bflr+</td>
<td>bflr-</td>
<td>bflr+</td>
<td>bflr-</td>
</tr>
<tr>
<td>bge+</td>
<td>bge-</td>
<td>bge+</td>
<td>bge-</td>
</tr>
<tr>
<td>bgectr+</td>
<td>bgectr-</td>
<td>bgectr+</td>
<td>bgectr-</td>
</tr>
<tr>
<td>bgel+</td>
<td>bgel-</td>
<td>bgel+</td>
<td>bgel-</td>
</tr>
<tr>
<td>bgelr+</td>
<td>bgelr-</td>
<td>bgelr+</td>
<td>bgelr-</td>
</tr>
<tr>
<td>bgtr+</td>
<td>bgta+</td>
<td>bgta-</td>
<td>bgtr-</td>
</tr>
<tr>
<td>bgtrctr+</td>
<td>bgtrctrl+</td>
<td>bgtrctr-</td>
<td>bgtrctrl-</td>
</tr>
<tr>
<td>bgtrl+</td>
<td>bgtla+</td>
<td>bgtla-</td>
<td>bgtrl-</td>
</tr>
<tr>
<td>bl+</td>
<td>bleo+</td>
<td>bleo-</td>
<td>bl-</td>
</tr>
<tr>
<td>blectr+</td>
<td>blectr-</td>
<td>blectr+</td>
<td>blectr-</td>
</tr>
<tr>
<td>blr+</td>
<td>blrlr+</td>
<td>blrlr-</td>
<td>blr-</td>
</tr>
<tr>
<td>bne+</td>
<td>bnea+</td>
<td>bnea-</td>
<td>bne-</td>
</tr>
<tr>
<td>bnectr+</td>
<td>bnectr-</td>
<td>bnectr+</td>
<td>bnectr-</td>
</tr>
<tr>
<td>bnl+</td>
<td>bnel+</td>
<td>bnela+</td>
<td>bnl-</td>
</tr>
<tr>
<td>bnlr+</td>
<td>bnlrlr+</td>
<td>bnlrlr-</td>
<td>bnlr-</td>
</tr>
<tr>
<td>bng+</td>
<td>bng+</td>
<td>bng+</td>
<td>bng-</td>
</tr>
<tr>
<td>bngctr+</td>
<td>bngctr-</td>
<td>bngctr+</td>
<td>bngctr-</td>
</tr>
<tr>
<td>bngl+</td>
<td>bngl+</td>
<td>bngla+</td>
<td>bngl-</td>
</tr>
<tr>
<td>bnglr+</td>
<td>bnglr+</td>
<td>bnglr+</td>
<td>bnglr-</td>
</tr>
<tr>
<td>bn+</td>
<td>bnl+</td>
<td>bnl+</td>
<td>bnl-</td>
</tr>
<tr>
<td>bnltc+</td>
<td>bnlctr+</td>
<td>bnlctr-</td>
<td>bnlct+</td>
</tr>
<tr>
<td>bnlr+</td>
<td>bnlrlr+</td>
<td>bnlrlr-</td>
<td>bnlr-</td>
</tr>
<tr>
<td>bns+</td>
<td>bns+</td>
<td>bns+</td>
<td>bns-</td>
</tr>
<tr>
<td>bnsctr+</td>
<td>bnsctr-</td>
<td>bnsctr+</td>
<td>bnsctr-</td>
</tr>
<tr>
<td>bns+</td>
<td>bns+</td>
<td>bns+</td>
<td>bns-</td>
</tr>
<tr>
<td>bnsl+</td>
<td>bns+</td>
<td>bns+</td>
<td>bns-</td>
</tr>
<tr>
<td>bnsr+</td>
<td>bnsrl+</td>
<td>bnsrl-</td>
<td>bnsr-</td>
</tr>
<tr>
<td>bnu+</td>
<td>bnu+</td>
<td>bnu+</td>
<td>bnu-</td>
</tr>
<tr>
<td>bnuctr+</td>
<td>bnuctr-</td>
<td>bnuctr+</td>
<td>bnuctr-</td>
</tr>
<tr>
<td>bnl+</td>
<td>bnl+</td>
<td>bnl+</td>
<td>bnl-</td>
</tr>
<tr>
<td>bnlr+</td>
<td>bnlrlr+</td>
<td>bnlrlr-</td>
<td>bnlr-</td>
</tr>
<tr>
<td>bnz+</td>
<td>bnz+</td>
<td>bnz+</td>
<td>bnz-</td>
</tr>
<tr>
<td>bnzctr+</td>
<td>bnzctr-</td>
<td>bnzctr+</td>
<td>bnzctr-</td>
</tr>
<tr>
<td>bnzlr+</td>
<td>bnzlr+</td>
<td>bnzlr+</td>
<td>bnzlr-</td>
</tr>
<tr>
<td>bso+</td>
<td>bso+</td>
<td>bso+</td>
<td>bso-</td>
</tr>
<tr>
<td>bsoctr+</td>
<td>bsoctr-</td>
<td>bsoctr+</td>
<td>bsoctr-</td>
</tr>
<tr>
<td>bso+</td>
<td>bso+</td>
<td>bso+</td>
<td>bso-</td>
</tr>
<tr>
<td>bsr+</td>
<td>bsr+</td>
<td>bsr+</td>
<td>bsr-</td>
</tr>
<tr>
<td>btr+</td>
<td>btr+</td>
<td>btr+</td>
<td>btr-</td>
</tr>
<tr>
<td>btctrl+</td>
<td>btctrl-</td>
<td>btctrl+</td>
<td>btctrl-</td>
</tr>
<tr>
<td>btlr+</td>
<td>btlrlr+</td>
<td>btlrlr-</td>
<td>btlr-</td>
</tr>
<tr>
<td>buna+</td>
<td>buna+</td>
<td>buna+</td>
<td>buna-</td>
</tr>
<tr>
<td>bunc+</td>
<td>bunc+</td>
<td>bunc+</td>
<td>bunc-</td>
</tr>
<tr>
<td>burl+</td>
<td>burl+</td>
<td>burl+</td>
<td>burl-</td>
</tr>
<tr>
<td>bunr+</td>
<td>bunrlr+</td>
<td>bunrlr-</td>
<td>bunr-</td>
</tr>
<tr>
<td>bza+</td>
<td>bza+</td>
<td>bza+</td>
<td>bza-</td>
</tr>
</tbody>
</table>

Chapter 6. Extended Instruction Mnemonics 95
Extended Mnemonics of Condition Register Logical Instructions

Extended mnemonics of condition register logical instructions are available in POWER family and PowerPC. These extended mnemonics are in the "com" assembly mode. Condition register logical instructions can be used to perform the following operations on a given condition register bit.

- Set bit to 1.
- Clear bit to 0.
- Copy bit.
- Invert bit.

The extended mnemonics shown in the following table allow these operations to be easily coded.

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>crset bx</td>
<td>creqv bx, bx, bx</td>
<td>Condition register set</td>
</tr>
<tr>
<td>crclr bx</td>
<td>crxor bx, bx, bx</td>
<td>Condition register clear</td>
</tr>
<tr>
<td>crmove bx, by</td>
<td>cror bx, by, by</td>
<td>Condition register move</td>
</tr>
<tr>
<td>crnot bx, by</td>
<td>crnor bx, by, by</td>
<td>Condition register NOT</td>
</tr>
</tbody>
</table>

Since the condition register logical instructions perform the operation on the condition register bit, the assembler supports expressions in all input operands. When using a symbol name to indicate a condition register (CR) field, the symbol name should be multiplied by four to get the correct CR bit, because each CR field has four bits.
**Examples**

1. To clear the SO bit (bit 3) of CR0:
   
   ```
   crclr so
   ```
   
   This is equivalent to:
   
   ```
   cpxor 3, 3, 3
   ```

2. To clear the EQ bit of CR3:
   
   ```
   crclr 4*cr3+eq
   ```
   
   This is equivalent to:
   
   ```
   cpxor 14, 14, 14
   ```

3. To invert the EQ bit of CR4 and place the result in the SO bit of CR5:

   ```
   crnot 4*cr5+so, 4*cr4+eq
   ```

   This is equivalent to:
   
   ```
   crnor 23, 18, 18
   ```

**Related Information**

- Chapter 6, “Extended Instruction Mnemonics,” on page 89.

- “Extended Mnemonics of Branch Instructions” on page 89.

- “Extended Mnemonics of Fixed-Point Arithmetic Instructions.”

- “Extended Mnemonics of Fixed-Point Compare Instructions” on page 98.

- “Extended Mnemonics of Fixed-Point Load Instructions” on page 99.

- “Extended Mnemonics of Fixed-Point Logical Instructions” on page 100.

- “Extended Mnemonics of Fixed-Point Trap Instructions” on page 100.

- “Extended Mnemonics of Moving from or to Special-Purpose Registers” on page 102.

- “Extended Mnemonics of 32-bit Fixed-Point Rotate and Shift Instructions” on page 107.

- “creqv (Condition Register Equivalent) Instruction” on page 165, “cror (Condition Register OR) Instruction” on page 168, “crnor (Condition Register NOR) Instruction” on page 167, “crxor (Condition Register XOR) Instruction” on page 170.

**Extended Mnemonics of Fixed-Point Arithmetic Instructions**

The following table shows the extended mnemonics for fixed-point arithmetic instructions for POWER family and PowerPC. Except as noted, these extended mnemonics are for POWER family and PowerPC and are in the com assembly mode.

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>subi rx, ry, value</td>
<td>addi rx, ry, -value</td>
<td>Subtract Immediate</td>
</tr>
<tr>
<td>subis rx, ry, value</td>
<td>addis rx, ry, -value</td>
<td>Subtract Immediate Shifted</td>
</tr>
<tr>
<td>subic[.], rx, ry, value</td>
<td>addic[.], rx, ry, -value</td>
<td>Subtract Immediate</td>
</tr>
<tr>
<td>subc[o][.], rx, ry, rz</td>
<td>subfc[o][.], rx, ry, rz</td>
<td>Subtract From Carrying</td>
</tr>
</tbody>
</table>

---

Chapter 6. Extended Instruction Mnemonics 97
Table 11. Fixed-Point Arithmetic Instruction Extended Mnemonics (continued)

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>si[,] rt, ra, value</td>
<td>ai[,] rt, ra, -value</td>
<td>Subtract Immediate</td>
</tr>
<tr>
<td>sub[0][] rx, ry, rz</td>
<td>subf[0][] rx, ry, rz</td>
<td>Subtract From</td>
</tr>
</tbody>
</table>

Note: The sub[0][] extended mnemonic is for PowerPC, since its base mnemonic subf[0][] is for PowerPC only.

Related Information

- Chapter 6, “Extended Instruction Mnemonics,” on page 89.
- “Extended Mnemonics of Branch Instructions” on page 89.
- “Extended Mnemonics of Condition Register Logical Instructions” on page 96.
- “Extended Mnemonics of Fixed-Point Compare Instructions.”
- “Extended Mnemonics of Fixed-Point Load Instructions” on page 99.
- “Extended Mnemonics of Fixed-Point Logical Instructions” on page 100.
- “Extended Mnemonics of Fixed-Point Trap Instructions” on page 100.
- “Extended Mnemonics of Moving from or to Special-Purpose Registers” on page 102.
- “Extended Mnemonics of 32-bit Fixed-Point Rotate and Shift Instructions” on page 107.

“addic or ai (Add Immediate Carrying) Instruction” on page 131, “addic. or ai. (Add Immediate Carrying and Record) Instruction” on page 132.

Extended Mnemonics of Fixed-Point Compare Instructions

The extended mnemonics for fixed-point compare instructions are shown in the following table. The input format of operands are different for POWER family and PowerPC. The L field for PowerPC supports 64-bit implementations. This field must have a value of 0 for 32-bit implementations. Since the POWER family architecture supports only 32-bit implementations, this field does not exist in POWER family. The assembler ensures that this bit is set to 0 for POWER family implementations. These extended mnemonics are in the com assembly mode.

Table 12. Fixed-Point Compare Instruction Extended Mnemonics

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpdi ra, value</td>
<td>cmpi 0, 1, ra, value</td>
<td>Compare Word Immediate</td>
</tr>
<tr>
<td>cmpwi bf, ra, si</td>
<td>cmpi bf, 0, ra, si</td>
<td>Compare Word Immediate</td>
</tr>
<tr>
<td>cmpd ra, rb</td>
<td>cmp 0, 1, ra, rb</td>
<td>Compare Word</td>
</tr>
<tr>
<td>cmpw bf, ra, rb</td>
<td>cmp bf, 0, ra, rb</td>
<td>Compare Word</td>
</tr>
<tr>
<td>cmpldi ra, value</td>
<td>cmpli 0, 1, ra, value</td>
<td>Compare Logical Word Immediate</td>
</tr>
<tr>
<td>cmplwi bf, ra, ui</td>
<td>cmpli bf, 0, ra, ui</td>
<td>Compare Logical Word Immediate</td>
</tr>
<tr>
<td>cmpld ra, rb</td>
<td>cmpli 0, 1, ra, rb</td>
<td>Compare Logical Word</td>
</tr>
<tr>
<td>cmplw bf, ra, rb</td>
<td>cmpli bf, 0, ra, rb</td>
<td>Compare Logical Word</td>
</tr>
</tbody>
</table>
Related Information

Chapter 6, “Extended Instruction Mnemonics,” on page 89.

“Extended Mnemonics of Branch Instructions” on page 89.

“Extended Mnemonics of Condition Register Logical Instructions” on page 96.

“Extended Mnemonics of Fixed-Point Arithmetic Instructions” on page 97.

“Extended Mnemonics of Fixed-Point Load Instructions.”

“Extended Mnemonics of Fixed-Point Logical Instructions” on page 100.

“Extended Mnemonics of Fixed-Point Trap Instructions” on page 100.

“Extended Mnemonics of Moving from or to Special-Purpose Registers” on page 102.

“Extended Mnemonics of 32-bit Fixed-Point Rotate and Shift Instructions” on page 107.

“cmpi (Compare Immediate) Instruction” on page 157, “cmp (Compare) Instruction” on page 156, “cmpli (Compare Logical Immediate) Instruction” on page 160, “cmpl (Compare Logical) Instruction” on page 159.

Extended Mnemonics of Fixed-Point Load Instructions

The following table shows the extended mnemonics for fixed-point load instructions for POWER family and PowerPC. These extended mnemonics are in the com assembly mode.

Table 13. Fixed-Point Load Instruction Extended Mnemonics

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>li rx, value</td>
<td>addi rx, 0, value</td>
<td>Load Immediate</td>
</tr>
<tr>
<td>la rx, disp(ry)</td>
<td>addi rx, ry, disp</td>
<td>Load Address</td>
</tr>
<tr>
<td>lil rt, value</td>
<td>cal rt, value(0)</td>
<td>Load Immediate Lower</td>
</tr>
<tr>
<td>liu rt, value</td>
<td>cau rt, 0, value</td>
<td>Load Immediate Upper</td>
</tr>
<tr>
<td>lis rx, value</td>
<td>addis rx, 0, value</td>
<td>Load Immediate Shifted</td>
</tr>
</tbody>
</table>

Related Information

Chapter 6, “Extended Instruction Mnemonics,” on page 89.

“Extended Mnemonics of Branch Instructions” on page 89.

“Extended Mnemonics of Condition Register Logical Instructions” on page 96.

“Extended Mnemonics of Fixed-Point Arithmetic Instructions” on page 97.

“Extended Mnemonics of Fixed-Point Compare Instructions” on page 98.

“Extended Mnemonics of Fixed-Point Logical Instructions” on page 100.

“Extended Mnemonics of Fixed-Point Trap Instructions” on page 100.

“Extended Mnemonics of Moving from or to Special-Purpose Registers” on page 102.
Extended Mnemonics of Fixed-Point Logical Instructions

The extended mnemonics for fixed-point logical instructions are shown in the following table. These POWER family and PowerPC extended mnemonics are in the <code>com</code> assembly mode.

Table 14. Fixed-Point Logical Instruction Extended Mnemonics

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>ori 0, 0, 0</td>
<td>OR Immediate</td>
</tr>
<tr>
<td>mr[..] rx,ry</td>
<td>or[..] rx, ry, ry</td>
<td>OR</td>
</tr>
<tr>
<td>not[..] rx,ry</td>
<td>nor[..] rx, ry, ry</td>
<td>NOR</td>
</tr>
</tbody>
</table>

Related Information

Chapter 6, "Extended Instruction Mnemonics," on page 89.

"Extended Mnemonics of Branch Instructions" on page 89.

"Extended Mnemonics of Condition Register Logical Instructions" on page 96.

"Extended Mnemonics of Fixed-Point Arithmetic Instructions" on page 97.

"Extended Mnemonics of Fixed-Point Compare Instructions" on page 98.

"Extended Mnemonics of Fixed-Point Load Instructions" on page 99.

"Extended Mnemonics of Fixed-Point Trap Instructions."

"Extended Mnemonics of Moving from or to Special-Purpose Registers" on page 102.

"Extended Mnemonics of 32-bit Fixed-Point Rotate and Shift Instructions" on page 107.

"nor (NOR) Instruction" on page 333, "or (OR) Instruction" on page 334, "ori or oril (OR Immediate) Instruction" on page 336.

Extended Mnemonics of Fixed-Point Trap Instructions

The extended mnemonics for fixed-point trap instructions incorporate the most useful TO operand values. A standard set of codes, shown in the following table, has been adopted for the most common combinations of trap conditions. These extended mnemonics are in the <code>com</code> assembly mode.

Table 15. Fixed-Point Trap Instruction Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>TO Encoding</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt</td>
<td>10000</td>
<td>less than</td>
</tr>
<tr>
<td>le</td>
<td>10100</td>
<td>less than or equal</td>
</tr>
<tr>
<td>ng</td>
<td>10100</td>
<td>not greater than</td>
</tr>
<tr>
<td>eq</td>
<td>00100</td>
<td>equal</td>
</tr>
<tr>
<td>ge</td>
<td>01100</td>
<td>greater than or equal</td>
</tr>
</tbody>
</table>
Table 15. Fixed-Point Trap Instruction Codes (continued)

<table>
<thead>
<tr>
<th>Code</th>
<th>TO Encoding</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nl</td>
<td>01100</td>
<td>not less than</td>
</tr>
<tr>
<td>gt</td>
<td>01000</td>
<td>greater than</td>
</tr>
<tr>
<td>ne</td>
<td>11000</td>
<td>not equal</td>
</tr>
<tr>
<td>lt</td>
<td>00010</td>
<td>logically less than</td>
</tr>
<tr>
<td>le</td>
<td>00110</td>
<td>logically less than or equal</td>
</tr>
<tr>
<td>lg</td>
<td>00110</td>
<td>logically not greater than</td>
</tr>
<tr>
<td>lge</td>
<td>00101</td>
<td>logically greater than or equal</td>
</tr>
<tr>
<td>lnl</td>
<td>00101</td>
<td>logically not less than</td>
</tr>
<tr>
<td>lgt</td>
<td>00001</td>
<td>logically greater than</td>
</tr>
<tr>
<td>lne</td>
<td>00011</td>
<td>logically not equal</td>
</tr>
<tr>
<td>None</td>
<td>11111</td>
<td>Unconditional</td>
</tr>
</tbody>
</table>

The POWER family extended mnemonics for fixed-point trap instructions have the following format:
- \texttt{txx} or \texttt{txxi}

where \texttt{xx} is one of the codes specified in the preceding table.

The 64-bit PowerPC extended mnemonics for double-word, fixed-point trap instructions have the following format:
- \texttt{tdxx} or \texttt{tdxxi}

The PowerPC extended mnemonics for fixed-point trap instructions have the following formats:
- \texttt{twxx} or \texttt{twxxi}

where \texttt{xx} is one of the codes specified in the preceding table.

The \texttt{trap} instruction is an unconditional trap:
- \texttt{trap}

Examples
1. To trap if R10 is less than R20:
   \texttt{tlt 10, 20}
   This is equivalent to:
   \texttt{t 16, 10, 20}
2. To trap if R4 is equal to 0x10:
   \texttt{teqi 4, 0x10}
   This is equivalent to:
   \texttt{ti 0x4, 4, 0x10}
3. To trap unconditionally:
   \texttt{trap}
   This is equivalent to:
   \texttt{tw 31, 0, 0}
4. To trap if RX is not equal to RY:
This is equivalent to:
```
twi 24, RX, RY
```

5. To trap if RX is logically greater than 0x7FF:
```
twlgti RX, 0x7FF
```

This is equivalent to:
```
twi 1, RX, 0x7FF
```

**Related Information**

- Chapter 6, “Extended Instruction Mnemonics,” on page 89.
- “Extended Mnemonics of Branch Instructions” on page 89.
- “Extended Mnemonics of Condition Register Logical Instructions” on page 96.
- “Extended Mnemonics of Fixed-Point Arithmetic Instructions” on page 97.
- “Extended Mnemonics of Fixed-Point Compare Instructions” on page 98.
- “Extended Mnemonics of Fixed-Point Load Instructions” on page 99.
- “Extended Mnemonics of Fixed-Point Logical Instructions” on page 100.
- “Extended Mnemonics of Moving from or to Special-Purpose Registers.”
- “Extended Mnemonics of 32-bit Fixed-Point Rotate and Shift Instructions” on page 107.
- “tw or t (Trap Word) Instruction” on page 456, “twi or ti (Trap Word Immediate) Instruction” on page 457.

**Extended Mnemonic mtcr for Moving to the Condition Register**

The mtcr (Move to Condition Register) extended mnemonic copies the contents of the low order 32 bits of a general purpose register (GPR) to the condition register using the same style as the mfcr instruction.

The extended mnemonic mtcr Rx is equivalent to the instruction mtcrf 0xFF, Rx.

This extended mnemonic is in the com assembly mode.

**Extended Mnemonics of Moving from or to Special-Purpose Registers**

This article discusses the following extended mnemonics:

- “mfspr Extended Mnemonics for POWER family” on page 103
- “mfspr Extended Mnemonics for PowerPC” on page 103
- “mfspr Extended Mnemonics for PowerPC 601 RISC Microprocessor” on page 106
- “mtspr Extended Mnemonics for POWER family” on page 103
- “mtspr Extended Mnemonics for PowerPC” on page 104
- “mtspr Extended Mnemonics for PowerPC 601 RISC Microprocessor” on page 106
### mfspr Extended Mnemonics for POWER family

**Table 16. mfspr Extended Mnemonics for POWER family**

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Privileged</th>
<th>SPR Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>mfxer rt</td>
<td>mfspr rt,1</td>
<td>no</td>
<td>XER</td>
</tr>
<tr>
<td>mflr rt</td>
<td>mfspr rt,8</td>
<td>no</td>
<td>LR</td>
</tr>
<tr>
<td>mfctr rt</td>
<td>mfspr rt,9</td>
<td>no</td>
<td>CTR</td>
</tr>
<tr>
<td>mfmq rt</td>
<td>mfspr rt,0</td>
<td>no</td>
<td>MQ</td>
</tr>
<tr>
<td>mfrtcu rt</td>
<td>mfspr rt,4</td>
<td>no</td>
<td>RTCU</td>
</tr>
<tr>
<td>mfrtc1 rt</td>
<td>mfspr rt,5</td>
<td>no</td>
<td>RTCL</td>
</tr>
<tr>
<td>mfdec rt</td>
<td>mfspr rt,6</td>
<td>no</td>
<td>DEC</td>
</tr>
<tr>
<td>mftid rt</td>
<td>mfspr rt,17</td>
<td>yes</td>
<td>TID</td>
</tr>
<tr>
<td>mfdsisr rt</td>
<td>mfspr rt,18</td>
<td>yes</td>
<td>DSISR</td>
</tr>
<tr>
<td>mfdar rt</td>
<td>mfspr rt,19</td>
<td>yes</td>
<td>DAR</td>
</tr>
<tr>
<td>mfdsdr0 rt</td>
<td>mfspr rt,24</td>
<td>yes</td>
<td>SDR0</td>
</tr>
<tr>
<td>mfdsdr1 rt</td>
<td>mfspr rt,25</td>
<td>yes</td>
<td>SDR1</td>
</tr>
<tr>
<td>mfsrr0 rt</td>
<td>mfspr rt,26</td>
<td>yes</td>
<td>SRR0</td>
</tr>
<tr>
<td>mfsrr1 rt</td>
<td>mfspr rt,27</td>
<td>yes</td>
<td>SRR1</td>
</tr>
</tbody>
</table>

### mtspr Extended Mnemonics for POWER family

**Table 17. mtspr Extended Mnemonics for POWER family**

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Privileged</th>
<th>SPR Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>mfxer rs</td>
<td>mtspr 1,rs</td>
<td>no</td>
<td>XER</td>
</tr>
<tr>
<td>mflr rs</td>
<td>mtspr 8,rs</td>
<td>no</td>
<td>LR</td>
</tr>
<tr>
<td>mtctr rs</td>
<td>mtspr 9,rs</td>
<td>no</td>
<td>CTR</td>
</tr>
<tr>
<td>mtmq rs</td>
<td>mtspr 0,rs</td>
<td>no</td>
<td>MQ</td>
</tr>
<tr>
<td>mtrtcu rs</td>
<td>mtspr 20,rs</td>
<td>yes</td>
<td>RTCU</td>
</tr>
<tr>
<td>mtrtc1 rs</td>
<td>mtspr 21,rs</td>
<td>yes</td>
<td>RTCL</td>
</tr>
<tr>
<td>mtdec rs</td>
<td>mtspr 22,rs</td>
<td>yes</td>
<td>DEC</td>
</tr>
<tr>
<td>mttid rs</td>
<td>mtspr 17,rs</td>
<td>yes</td>
<td>TID</td>
</tr>
<tr>
<td>mtdsisr rs</td>
<td>mtspr 18,rs</td>
<td>yes</td>
<td>DSISR</td>
</tr>
<tr>
<td>mtdar rs</td>
<td>mtspr 19,rs</td>
<td>yes</td>
<td>DAR</td>
</tr>
<tr>
<td>mtsdr0 rs</td>
<td>mtspr 24,rs</td>
<td>yes</td>
<td>SDR0</td>
</tr>
<tr>
<td>mtsdr1 rs</td>
<td>mtspr 25,rs</td>
<td>yes</td>
<td>SDR1</td>
</tr>
<tr>
<td>mtsrr0 rs</td>
<td>mtspr 26,rs</td>
<td>yes</td>
<td>SRR0</td>
</tr>
<tr>
<td>mtsrr1 rs</td>
<td>mtspr 27,rs</td>
<td>yes</td>
<td>SRR1</td>
</tr>
</tbody>
</table>

### mfspr Extended Mnemonics for PowerPC

**Table 18. mfspr Extended Mnemonics for PowerPC**

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Privileged</th>
<th>SPR Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>mfxer rt</td>
<td>mfspr rt,1</td>
<td>no</td>
<td>XER</td>
</tr>
</tbody>
</table>
### Table 18. mfspr Extended Mnemonics for PowerPC (continued)

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Privileged</th>
<th>SPR Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>mflr rt</td>
<td>mfspr rt,8</td>
<td>no</td>
<td>LR</td>
</tr>
<tr>
<td>mfctr rt</td>
<td>mfspr rt,9</td>
<td>no</td>
<td>CTR</td>
</tr>
<tr>
<td>mfdsisr rt</td>
<td>mfspr rt,18</td>
<td>yes</td>
<td>DSISR</td>
</tr>
<tr>
<td>mfdrar rt</td>
<td>mfspr rt,19</td>
<td>yes</td>
<td>DAR</td>
</tr>
<tr>
<td>mfdec rt</td>
<td>mfspr rt,22</td>
<td>yes</td>
<td>DEC</td>
</tr>
<tr>
<td>mfsdr1 rt</td>
<td>mfspr rt,25</td>
<td>yes</td>
<td>SDR1</td>
</tr>
<tr>
<td>mfsrr0 r</td>
<td>mfspr rt,26</td>
<td>yes</td>
<td>SRR0</td>
</tr>
<tr>
<td>mfsrr1 rt</td>
<td>mfspr rt,27</td>
<td>yes</td>
<td>SRR1</td>
</tr>
<tr>
<td>mfsprg rt,0</td>
<td>mfspr rt,272</td>
<td>yes</td>
<td>SPRG0</td>
</tr>
<tr>
<td>mfsprg rt,1</td>
<td>mfspr rt,273</td>
<td>yes</td>
<td>SPRG1</td>
</tr>
<tr>
<td>mfsprg rt,2</td>
<td>mfspr rt,274</td>
<td>yes</td>
<td>SPRG2</td>
</tr>
<tr>
<td>mfsprg rt,3</td>
<td>mfspr rt,275</td>
<td>yes</td>
<td>SPRG3</td>
</tr>
<tr>
<td>mfear rt</td>
<td>mfspr rt,282</td>
<td>yes</td>
<td>EAR</td>
</tr>
<tr>
<td>mfpvr rt</td>
<td>mfspr rt,287</td>
<td>yes</td>
<td>PVR</td>
</tr>
<tr>
<td>mfibatu rt,0</td>
<td>mfspr rt,528</td>
<td>yes</td>
<td>IBAT0U</td>
</tr>
<tr>
<td>mfibati rt,1</td>
<td>mfspr rt,529</td>
<td>yes</td>
<td>IBAT0L</td>
</tr>
<tr>
<td>mfibati rt,1</td>
<td>mfspr rt,530</td>
<td>yes</td>
<td>IBAT1U</td>
</tr>
<tr>
<td>mfibati rt,1</td>
<td>mfspr rt,531</td>
<td>yes</td>
<td>IBAT1L</td>
</tr>
<tr>
<td>mfibati rt,2</td>
<td>mfspr rt,532</td>
<td>yes</td>
<td>IBAT2U</td>
</tr>
<tr>
<td>mfibati rt,2</td>
<td>mfspr rt,533</td>
<td>yes</td>
<td>IBAT2L</td>
</tr>
<tr>
<td>mfibati rt,3</td>
<td>mfspr rt,534</td>
<td>yes</td>
<td>IBAT3U</td>
</tr>
<tr>
<td>mfibati rt,3</td>
<td>mfspr rt,535</td>
<td>yes</td>
<td>IBAT3L</td>
</tr>
<tr>
<td>mfdbatu rt,0</td>
<td>mfspr rt,536</td>
<td>yes</td>
<td>DBAT0U</td>
</tr>
<tr>
<td>mfdbati rt,0</td>
<td>mfspr rt,537</td>
<td>yes</td>
<td>DBAT0L</td>
</tr>
<tr>
<td>mfdbatu rt,1</td>
<td>mfspr rt,538</td>
<td>yes</td>
<td>DBAT1U</td>
</tr>
<tr>
<td>mfdbati rt,1</td>
<td>mfspr rt,539</td>
<td>yes</td>
<td>DBAT1L</td>
</tr>
<tr>
<td>mfdbatu rt,2</td>
<td>mfspr rt,540</td>
<td>yes</td>
<td>DBAT2U</td>
</tr>
<tr>
<td>mfdbati rt,2</td>
<td>mfspr rt,541</td>
<td>yes</td>
<td>DBAT2L</td>
</tr>
<tr>
<td>mfdbatu rt,3</td>
<td>mfspr rt,542</td>
<td>yes</td>
<td>DBAT3U</td>
</tr>
<tr>
<td>mfdbati rt,3</td>
<td>mfspr rt,543</td>
<td>yes</td>
<td>DBAT3L</td>
</tr>
</tbody>
</table>

**Note:** The `mfdec` instruction is a privileged instruction in PowerPC. The encoding for this instruction in PowerPC differs from that in POWER family. See the "mfspr (Move from Special-Purpose Register) Instruction" on page 303 for information on this instruction. "Differences between POWER family and PowerPC Instructions with the Same Op Code" on page 115 provides a summary of the differences for this instruction for POWER family and PowerPC.

### mtspr Extended Mnemonics for PowerPC

**Table 19. mtspr Extended Mnemonics for PowerPC**

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Privileged</th>
<th>SPR Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtxer rs</td>
<td>mtspr 1,rs</td>
<td>no</td>
<td>XER</td>
</tr>
</tbody>
</table>
Table 19. mtspr Extended Mnemonics for PowerPC (continued)

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Privileged</th>
<th>SPR Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtlr rs</td>
<td>mtspr 8,rs</td>
<td>no</td>
<td>LR</td>
</tr>
<tr>
<td>mtctr rs</td>
<td>mtspr 9,rs</td>
<td>no</td>
<td>CTR</td>
</tr>
<tr>
<td>mtdsisr rs</td>
<td>mtspr 19,rs</td>
<td>yes</td>
<td>DSISR</td>
</tr>
<tr>
<td>mtdar rs</td>
<td>mtspr 19,rs</td>
<td>yes</td>
<td>DAR</td>
</tr>
<tr>
<td>mtdec rs</td>
<td>mtspr 22,rs</td>
<td>yes</td>
<td>DEC</td>
</tr>
<tr>
<td>mtsdr1 rs</td>
<td>mtspr 25,rs</td>
<td>yes</td>
<td>SDR1</td>
</tr>
<tr>
<td>mtsrr0 rs</td>
<td>mtspr 26,rs</td>
<td>yes</td>
<td>SRR0</td>
</tr>
<tr>
<td>mtsrr1 rs</td>
<td>mtspr 27,rs</td>
<td>yes</td>
<td>SRR1</td>
</tr>
<tr>
<td>mtspreg 0,rs</td>
<td>mtspr 272,rs</td>
<td>yes</td>
<td>SPRG0</td>
</tr>
<tr>
<td>mtspreg 1,rs</td>
<td>mtspr 273,rs</td>
<td>yes</td>
<td>SPRG1</td>
</tr>
<tr>
<td>mtspreg 2,rs</td>
<td>mtspr 274,rs</td>
<td>yes</td>
<td>SPRG2</td>
</tr>
<tr>
<td>mtspreg 3,rs</td>
<td>mtspr 275,rs</td>
<td>yes</td>
<td>SPRG3</td>
</tr>
<tr>
<td>mttear rs</td>
<td>mtspr 282,rs</td>
<td>yes</td>
<td>EAR</td>
</tr>
<tr>
<td>mttbl rs (or mttb rs)</td>
<td>mtspr 284,rs</td>
<td>yes</td>
<td>TBL</td>
</tr>
<tr>
<td>mt_tbu rs</td>
<td>mtspr 285,rs</td>
<td>yes</td>
<td>TBU</td>
</tr>
<tr>
<td>mtibatu 0,rs</td>
<td>mtspr 528,rs</td>
<td>yes</td>
<td>IBAT0U</td>
</tr>
<tr>
<td>mtibatI 0,rs</td>
<td>mtspr 529,rs</td>
<td>yes</td>
<td>IBAT0L</td>
</tr>
<tr>
<td>mtibatu 1,rs</td>
<td>mtspr 530,rs</td>
<td>yes</td>
<td>IBAT1U</td>
</tr>
<tr>
<td>mtibatI 1,rs</td>
<td>mtspr 531,rs</td>
<td>yes</td>
<td>IBAT1L</td>
</tr>
<tr>
<td>mtibatu 2,rs</td>
<td>mtspr 532,rs</td>
<td>yes</td>
<td>IBAT2U</td>
</tr>
<tr>
<td>mtibatI 2,rs</td>
<td>mtspr 533,rs</td>
<td>yes</td>
<td>IBAT2L</td>
</tr>
<tr>
<td>mtibatu 3,rs</td>
<td>mtspr 534,rs</td>
<td>yes</td>
<td>IBAT3U</td>
</tr>
<tr>
<td>mtibatI 3,rs</td>
<td>mtspr 535,rs</td>
<td>yes</td>
<td>IBAT3L</td>
</tr>
<tr>
<td>mt_dbatu 0,rs</td>
<td>mtspr 536,rs</td>
<td>yes</td>
<td>DBAT0U</td>
</tr>
<tr>
<td>mt_dbatI 0,rs</td>
<td>mtspr 537,rs</td>
<td>yes</td>
<td>DBAT0L</td>
</tr>
<tr>
<td>mt_dbatu 1,rs</td>
<td>mtspr 538,rs</td>
<td>yes</td>
<td>DBAT1U</td>
</tr>
<tr>
<td>mt_dbatI 1,rs</td>
<td>mtspr 539,rs</td>
<td>yes</td>
<td>DBAT1L</td>
</tr>
<tr>
<td>mt_dbatu 2,rs</td>
<td>mtspr 540,rs</td>
<td>yes</td>
<td>DBAT2U</td>
</tr>
<tr>
<td>mt_dbatI 2,rs</td>
<td>mtspr 541,rs</td>
<td>yes</td>
<td>DBAT2L</td>
</tr>
<tr>
<td>mt_dbatI 3,rs</td>
<td>mtspr 542,rs</td>
<td>yes</td>
<td>DBAT3U</td>
</tr>
<tr>
<td>mt_dbatI 3,rs</td>
<td>mtspr 543,rs</td>
<td>yes</td>
<td>DBAT3L</td>
</tr>
</tbody>
</table>

**Note:** The **mfdec** instruction is a privileged instruction in PowerPC. The encoding for this instruction in PowerPC differs from that in POWER family. See the ["mfspr (Move from Special-Purpose Register) Instruction" on page 303](#) for information on this instruction. "Differences between POWER family and PowerPC Instructions with the Same Op Code" on page 115 provides a summary of the differences for this instruction for POWER family and PowerPC.
### mfspr Extended Mnemonics for PowerPC 601 RISC Microprocessor

**Table 20. mfspr Extended Mnemonics for PowerPC 601 RISC Microprocessor**

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Privileged</th>
<th>SPR Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>mfmq rt</td>
<td>mfspr rt,0</td>
<td>no</td>
<td>MQ</td>
</tr>
<tr>
<td>mfxer rt</td>
<td>mfspr rt,1</td>
<td>no</td>
<td>XER</td>
</tr>
<tr>
<td>mfrcru rt</td>
<td>mfspr rt,4</td>
<td>no</td>
<td>RTCU</td>
</tr>
<tr>
<td>mfrcu1 rt</td>
<td>mfspr rt,5</td>
<td>no</td>
<td>RTCL</td>
</tr>
<tr>
<td>mfdec rt</td>
<td>mfspr rt,6</td>
<td>no</td>
<td>DEC</td>
</tr>
<tr>
<td>mfllr rt</td>
<td>mfspr rt,8</td>
<td>no</td>
<td>LR</td>
</tr>
<tr>
<td>mfcrt r</td>
<td>mfspr rt,9</td>
<td>no</td>
<td>CTR</td>
</tr>
<tr>
<td>mfsdsi r</td>
<td>mfspr rt,18</td>
<td>yes</td>
<td>DSISR</td>
</tr>
<tr>
<td>mfsdai r</td>
<td>mfspr rt,19</td>
<td>yes</td>
<td>DAR</td>
</tr>
<tr>
<td>mfsdr1 rt</td>
<td>mfspr rt,25</td>
<td>yes</td>
<td>SDR1</td>
</tr>
<tr>
<td>mfsdr0 r</td>
<td>mfspr rt,26</td>
<td>yes</td>
<td>SRR0</td>
</tr>
<tr>
<td>mfsdr1 r</td>
<td>mfspr rt,27</td>
<td>yes</td>
<td>SRR1</td>
</tr>
<tr>
<td>mfsprg rt,0</td>
<td>mfspr rt,272</td>
<td>yes</td>
<td>SPRG0</td>
</tr>
<tr>
<td>mfsprg rt,1</td>
<td>mfspr rt,273</td>
<td>yes</td>
<td>SPRG1</td>
</tr>
<tr>
<td>mfsprg rt,2</td>
<td>mfspr rt,274</td>
<td>yes</td>
<td>SPRG2</td>
</tr>
<tr>
<td>mfsprg rt,3</td>
<td>mfspr rt,275</td>
<td>yes</td>
<td>SPRG3</td>
</tr>
<tr>
<td>mfsprg rt</td>
<td>mfspr rt,282</td>
<td>yes</td>
<td>EAR</td>
</tr>
<tr>
<td>mfsprg rt</td>
<td>mfspr rt,287</td>
<td>yes</td>
<td>PVR</td>
</tr>
</tbody>
</table>

### mtspr Extended Mnemonics for PowerPC 601 RISC Microprocessor

**Table 21. mtspr Extended Mnemonics for PowerPC 601 RISC Microprocessor**

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Privileged</th>
<th>SPR Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtmq rs</td>
<td>mtspr 0,rs</td>
<td>no</td>
<td>MQ</td>
</tr>
<tr>
<td>mtxer rs</td>
<td>mtspr 1,rs</td>
<td>no</td>
<td>XER</td>
</tr>
<tr>
<td>mtllr rs</td>
<td>mtspr 8,rs</td>
<td>no</td>
<td>LR</td>
</tr>
<tr>
<td>mtcrt r</td>
<td>mtspr 9,rs</td>
<td>no</td>
<td>CTR</td>
</tr>
<tr>
<td>mtldsir rs</td>
<td>mtspr 18,rs</td>
<td>yes</td>
<td>DSISR</td>
</tr>
<tr>
<td>mtdar rs</td>
<td>mtspr 19,rs</td>
<td>yes</td>
<td>DAR</td>
</tr>
<tr>
<td>mtclcr rs</td>
<td>mtspr 20,rs</td>
<td>yes</td>
<td>RTCU</td>
</tr>
<tr>
<td>mtclcl r</td>
<td>mtspr 21,rs</td>
<td>yes</td>
<td>RTCL</td>
</tr>
<tr>
<td>mtdec rs</td>
<td>mtspr 22,rs</td>
<td>yes</td>
<td>DEC</td>
</tr>
<tr>
<td>mttsdr1 rs</td>
<td>mtspr 25,rs</td>
<td>yes</td>
<td>SDR1</td>
</tr>
<tr>
<td>mttsr0 rs</td>
<td>mtspr 26,rs</td>
<td>yes</td>
<td>SRR0</td>
</tr>
<tr>
<td>mttsrr1 rs</td>
<td>mtspr 27,rs</td>
<td>yes</td>
<td>SRR1</td>
</tr>
<tr>
<td>mtsprr 0,rs</td>
<td>mtspr 272,rs</td>
<td>yes</td>
<td>SPRG0</td>
</tr>
<tr>
<td>mtsprr 1,rs</td>
<td>mtspr 273,rs</td>
<td>yes</td>
<td>SPRG1</td>
</tr>
<tr>
<td>mtsprr 2,rs</td>
<td>mtspr 274,rs</td>
<td>yes</td>
<td>SPRG2</td>
</tr>
<tr>
<td>mtsprr 3,rs</td>
<td>mtspr 275,rs</td>
<td>yes</td>
<td>SPRG3</td>
</tr>
</tbody>
</table>
Table 21. mtspr Extended Mnemonics for PowerPC 601 RISC Microprocessor (continued)

<table>
<thead>
<tr>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Privileged</th>
<th>SPR Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtear rs</td>
<td>mtspr 282,rs</td>
<td>yes</td>
<td>EAR</td>
</tr>
</tbody>
</table>

**Related Information**

- Chapter 6, “Extended Instruction Mnemonics,” on page 89.
- “Extended Mnemonics of Branch Instructions” on page 89.
- “Extended Mnemonics of Condition Register Logical Instructions” on page 96.
- “Extended Mnemonics of Fixed-Point Arithmetic Instructions” on page 97.
- “Extended Mnemonics of Fixed-Point Compare Instructions” on page 98.
- “Extended Mnemonics of Fixed-Point Load Instructions” on page 99.
- “Extended Mnemonics of Fixed-Point Logical Instructions” on page 100.
- “Extended Mnemonics of Fixed-Point Trap Instructions” on page 100.
- “Extended Mnemonics of 32-bit Fixed-Point Rotate and Shift Instructions.”

**Extended Mnemonics of 32-bit Fixed-Point Rotate and Shift Instructions**

A set of extended mnemonics are provided for extract, insert, rotate, shift, clear, and clear left and shift left operations. This article discusses the following:

- “Alternative Input Format”
- “32-bit Rotate and Shift Extended Mnemonics for POWER family and PowerPC” on page 108

**Alternative Input Format**

The alternative input format is applied to the following POWER family and PowerPC instructions.

<table>
<thead>
<tr>
<th>POWER family</th>
<th>PowerPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>rlimi[,]</td>
<td>rlimi[,]</td>
</tr>
<tr>
<td>rlinm[,]</td>
<td>rlinm[,]</td>
</tr>
<tr>
<td>rlinm[,]</td>
<td>rlinm[,]</td>
</tr>
<tr>
<td>rlimi[,]</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Five operands are normally required for these instructions. These operands are:

- RA, RS, SH, MB, ME

MB indicates the first bit with a value of 1 in the mask, and ME indicates the last bit with a value of 1 in the mask. The assembler supports the following operand format:

- RA, RS, SH, BM
BM is the mask itself. The assembler generates the MB and ME operands from the BM operand for the instructions. The assembler checks the BM operand first. If an invalid BM is entered, error 78 is reported.

A valid mask is defined as a single series (one or more) of bits with a value of 1 surrounded by zero or more bits with a value of 0. A mask of all bits with a value of 0 may not be specified.

**Examples of Valid 32-bit Masks**

The following shows examples of valid 32-bit masks.

<table>
<thead>
<tr>
<th>BM</th>
<th>ME</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>00000000000000000000011111111111</td>
</tr>
<tr>
<td>12</td>
<td>25</td>
<td>00000000000000000000011111111111</td>
</tr>
</tbody>
</table>

**Examples of 32-bit Masks That Are Not Valid**

The following shows examples of 32-bit masks that are not valid.

<table>
<thead>
<tr>
<th>BM</th>
<th>ME</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11111111111111111111111111111111</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>00000000000000000000011111111111</td>
</tr>
</tbody>
</table>

**32-bit Rotate and Shift Extended Mnemonics for POWER family and PowerPC**

The extended mnemonics for the rotate and shift instructions are in the POWER family and PowerPC intersection area (com assembly mode). A set of rotate and shift extended mnemonics provide for the following operations:

- **Extract**
  - Selects a field of \( n \) bits starting at bit position \( b \) in the source register. This field is right- or left-justified in the target register. All other bits of the target register are cleared to 0.

- **Insert**
  - Selects a left- or right-justified field of \( n \) bits in the source register. This field is inserted starting at bit position \( b \) of the target register. Other bits of the target register are unchanged. No extended mnemonic is provided for insertion of a left-justified field when operating on doublewords, since such an insertion requires more than one instruction.

- **Rotate**
  - Rotates the contents of a register right or left \( n \) bits without masking.

- **Shift**
  - Shifts the contents of a register right or left \( n \) bits. Vacated bits are cleared to 0 (logical shift).

- **Clear**
  - Clears the leftmost or rightmost \( n \) bits of a register to 0.

- **Clear left and shift left**
  - Clears the leftmost \( b \) bits of a register, then shifts the register by \( n \) bits. This operation can be used to scale a known nonnegative array index by the width of an element.

The rotate and shift extended mnemonics are shown in the following table. The \( N \) operand specifies the number of bits to be extracted, inserted, rotated, or shifted. Because expressions are introduced when the extended mnemonics are mapped to the base mnemonics, certain restrictions are imposed to prevent the result of the expression from causing an overflow in the \( SH, MB, \) or \( ME \) operand.
To maintain compatibility with previous versions of AIX, \( n \) is not restricted to a value of 0. If \( n \) is 0, the assembler treats \( 32-n \) as a value of 0.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract and left justify immediate</td>
<td><code>extlwi RA, RS, n, b</code></td>
<td><code>rlwinm RA, RS, b, 0, n-1</code></td>
<td>( 32 &gt; n &gt; 0 )</td>
</tr>
<tr>
<td>Extract and right justify immediate</td>
<td><code>extrwi RA, RS, n, b</code></td>
<td><code>rlwinm RA, RS, b+n, 32-n, 31</code></td>
<td>( 32 &gt; n &gt; 0 ) &amp; ( b+n \leq 32 )</td>
</tr>
<tr>
<td>Insert from left immediate</td>
<td><code>inlwi RA, RS, n, b</code></td>
<td><code>rlwinm RA, RS, 32-b, b, (b+n)-1</code></td>
<td>( b+n &lt;= 32 ) &amp; ( 32 &gt; n &gt; 0 ) &amp; ( b &gt; b &gt;= 0 )</td>
</tr>
<tr>
<td>Insert from right immediate</td>
<td><code>insrwi RA, RS, n, b</code></td>
<td><code>rlwinm RA, RS, 32-(b+n), b, (b+n)-1</code></td>
<td>( b+n &lt;= 32 ) &amp; ( 32 &gt; n &gt; 0 )</td>
</tr>
<tr>
<td>Rotate left immediate</td>
<td><code>rotlwi RA, RS, n</code></td>
<td><code>rlwinm RA, RS, n, 0, 31</code></td>
<td>( 32 &gt; n &gt;= 0 )</td>
</tr>
<tr>
<td>Rotate right immediate</td>
<td><code>rotrwi RA, RS, n</code></td>
<td><code>rlwinm RA, RS, 32-n, 0, 31</code></td>
<td>( 32 &gt; n &gt;= 0 )</td>
</tr>
<tr>
<td>Rotate left</td>
<td><code>rotw RA, RS, b</code></td>
<td><code>rlwinm RA, RS, RB, 0, 31</code></td>
<td>None</td>
</tr>
<tr>
<td>Shift left immediate</td>
<td><code>slwi RA, RS, n</code></td>
<td><code>rlwinm RA, RS, n, 0, 31-n</code></td>
<td>( 32 &gt; n &gt;= 0 )</td>
</tr>
<tr>
<td>Shift right immediate</td>
<td><code>srwi RA, RS, n</code></td>
<td><code>rlwinm RA, RS, 32-n, 0, 31</code></td>
<td>( 32 &gt; n &gt;= 0 )</td>
</tr>
<tr>
<td>Clear left immediate</td>
<td><code>clrlwi RA, RS, n</code></td>
<td><code>rlwinm RA, RS, 0, 0, 31</code></td>
<td>( 32 &gt; n &gt;= 0 )</td>
</tr>
<tr>
<td>Clear right immediate</td>
<td><code>clrrwi RA, RS, n</code></td>
<td><code>rlwinm RA, RS, 0, 0, 31-n</code></td>
<td>( 32 &gt; n &gt;= 0 )</td>
</tr>
<tr>
<td>Clear left and shift left immediate</td>
<td><code>clrslwi RA, RS, b, n</code></td>
<td><code>rlwinm RA, RS, b-n, 31-n</code></td>
<td>( b+n &gt;= 0 ) &amp; ( 32 &gt; n &gt;= 0 ) &amp; ( 32 &gt; b &gt;= 0 )</td>
</tr>
</tbody>
</table>

Notes:
1. In POWER family, the mnemonic `slwi[.]` is `sli[.]`. The mnemonic `srwi[.]` is `sri[.]`.
2. All of these extended mnemonics can be coded with a final . (period) to cause the Rc bit to be set in the underlying instruction.

Examples
1. To extract the sign bit (bit 31) of register \( RY \) and place the result right-justified into register \( RX \):
   ```
   extrwi RX, RY, 1, 0
   ```
   This is equivalent to:
   ```
   rlwinm RX, RY, 1, 31, 31
   ```
2. To insert the bit extracted in Example 1 into the sign bit (bit 31) of register \( RX \):
   ```
   insrwi RZ, RX, 1, 0
   ```
   This is equivalent to:
   ```
   rlwimi RZ, RX, 31, 0, 0
   ```
3. To shift the contents of register \( RX \) left 8 bits and clear the high-order 32 bits:
   ```
   slwi RX, RX, 8
   ```
   This is equivalent to:
   ```
   rlwinm RX, RX, 8, 0, 23
   ```
4. To clear the high-order 16 bits of the low-order 32 bits of register \( RY \) and place the result in register \( RX \), and clear the high-order 32 bits of register \( RX \):
   ```
   clrlwi RX, RX, 16
   ```
   This is equivalent to:
Related Information
Chapter 6, “Extended Instruction Mnemonics,” on page 89.
“Extended Mnemonics of Branch Instructions” on page 89.
“Extended Mnemonics of Condition Register Logical Instructions” on page 96.
“Extended Mnemonics of Fixed-Point Arithmetic Instructions” on page 97.
“Extended Mnemonics of Fixed-Point Compare Instructions” on page 98.
“Extended Mnemonics of Fixed-Point Load Instructions” on page 99.
“Extended Mnemonics of Fixed-Point Logical Instructions” on page 100.
“Extended Mnemonics of Fixed-Point Trap Instructions” on page 100.
“Extended Mnemonics of Moving from or to Special-Purpose Registers” on page 102.

“addic or ai (Add Immediate Carrying) Instruction” on page 131.
“addic. or ai. (Add Immediate Carrying and Record) Instruction” on page 132.
“bc (Branch Conditional) Instruction” on page 144.
“bclr or bcr (Branch Conditional Link Register) Instruction” on page 149.
“bcctr or bcc (Branch Conditional to Count Register) Instruction” on page 147.
“addi (Add Immediate) or cal (Compute Address Lower) Instruction” on page 130.
“addis or cau (Add Immediate Shifted) Instruction” on page 133.
“cmpi (Compare Immediate) Instruction” on page 157.
“cmp (Compare) Instruction” on page 156.
“cmpi (Compare Logical Immediate) Instruction” on page 156.
“cmp (Compare Logical) Instruction” on page 159.
“creqv (Condition Register Equivalent) Instruction” on page 165.
“cror (Condition Register OR) Instruction” on page 168.
“crnor (Condition Register NOR) Instruction” on page 167.
“crxor (Condition Register XOR) Instruction” on page 170.
“mfspr (Move from Special-Purpose Register) Instruction” on page 303.
“mtspr (Move to Special-Purpose Register) Instruction” on page 303.
“rlimi” on page 335.
“rlwinm or rlwinm (Rotate Left Word Immediate Then AND with Mask) Instruction” on page 354.
“tw or ti (Trap Word) Instruction” on page 456.
“twi or ti (Trap Word Immediate) Instruction” on page 457.

Extended Mnemonics of 64-bit Fixed-Point Rotate and Shift Instructions
A set of extended mnemonics are provided for extract, insert, rotate, shift, clear, and clear left and shift left operations. This article discusses the following:

- “Alternative Input Format” on page 107
- “32-bit Rotate and Shift Extended Mnemonics for POWER family and PowerPC” on page 108

Alternative Input Format
The alternative input format is applied to the following POWER family and PowerPC instructions.

<table>
<thead>
<tr>
<th>POWER family</th>
<th>PowerPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>rlimi[,]</td>
<td>rlimi[,]</td>
</tr>
<tr>
<td>rlinm[,]</td>
<td>rlinm[,]</td>
</tr>
<tr>
<td>rlnm[,]</td>
<td>rlnm[,]</td>
</tr>
<tr>
<td>rlim[,]</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Five operands are normally required for these instructions. These operands are:
RA, RS, SH, MB, ME

MB indicates the first bit with a value of 1 in the mask, and ME indicates the last bit with a value of 1 in the mask. The assembler supports the following operand format.

RA, RS, SH, BM

BM is the mask itself. The assembler generates the MB and ME operands from the BM operand for the instructions. The assembler checks the BM operand first. If an invalid BM is entered, error 78 is reported.

A valid mask is defined as a single series (one or more) of bits with a value of 1 surrounded by zero or more bits with a value of 0. A mask of all bits with a value of 0 may not be specified.

64-bit Rotate and Shift Extended Mnemonics for POWER family and PowerPC

The extended mnemonics for the rotate and shift instructions are in the POWER family and PowerPC intersection area (com assembly mode). A set of rotate and shift extended mnemonics provide for the following operations:

Extract
Selects a field of \(n\) bits starting at bit position \(b\) in the source register. This field is right- or left-justified in the target register. All other bits of the target register are cleared to 0.

Insert
Selects a left- or right-justified field of \(n\) bits in the source register. This field is inserted starting at bit position \(b\) of the target register. Other bits of the target register are unchanged. No extended mnemonic is provided for insertion of a left-justified field when operating on doublewords, since such an insertion requires more than one instruction.

Rotate
Rotates the contents of a register right or left \(n\) bits without masking.

Shift
Shifts the contents of a register right or left \(n\) bits. Vacated bits are cleared to 0 (logical shift).

Clear
Clears the leftmost or rightmost \(n\) bits of a register to 0.

Clear left and shift left
Clears the leftmost \(b\) bits of a register, then shifts the register by \(n\) bits. This operation can be used to scale a known nonnegative array index by the width of an element.

The rotate and shift extended mnemonics are shown in the following table. The \(N\) operand specifies the number of bits to be extracted, inserted, rotated, or shifted. Because expressions are introduced when the extended mnemonics are mapped to the base mnemonics, certain restrictions are imposed to prevent the result of the expression from causing an overflow in the \(SH, MB,\) or \(ME\) operand.

To maintain compatibility with previous versions of AIX, \(n\) is not restricted to a value of 0. If \(n\) is 0, the assembler treats 32-\(n\) as a value of 0.

Table 23. 63-bit Rotate and Shift Extended Mnemonics for PowerPC

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract double word and right justify immediate</td>
<td>extrdi RA, RS, (n, b)</td>
<td>rldicl RA, RS, (b + n, 64 - n)</td>
<td>(n &gt; 0)</td>
</tr>
<tr>
<td>Rotate double word left immediate</td>
<td>rotdi RA, RS, (n)</td>
<td>rldicl RA, RS, (n, 0)</td>
<td>None</td>
</tr>
<tr>
<td>Rotate double word right immediate</td>
<td>rotdi RA, RS, (n)</td>
<td>rldicl RA, RS, (64 - n, 0)</td>
<td>None</td>
</tr>
<tr>
<td>Rotate double word right immediate</td>
<td>srdi RA, RS, (n)</td>
<td>rldicl RA, RS, (64 - n, n)</td>
<td>(n &lt; 64)</td>
</tr>
</tbody>
</table>
### Table 23. 63-bit Rotate and Shift Extended Mnemonics for PowerPC (continued)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear left double word immediate</td>
<td>clrldi RA, RS, n</td>
<td>rldicl RA, RS, 0, n</td>
<td>n &lt; 64</td>
</tr>
<tr>
<td>Extract double word and left justify immediate</td>
<td>extldi RA, RS, n, b</td>
<td>rldicr RA, RS, b, n - 1</td>
<td>None</td>
</tr>
<tr>
<td>Shift left double word immediate</td>
<td>sidi RA, RS, n</td>
<td>rldicr RA, RS, n, 63 - n</td>
<td>None</td>
</tr>
<tr>
<td>Clear right double word immediate</td>
<td>clrldi RA, RS, n</td>
<td>rldicr RA, RS, 0, 63 - n</td>
<td>None</td>
</tr>
<tr>
<td>Clear left double word and shift left immediate</td>
<td>clrlsldi RA, RS, b, n</td>
<td>rldic RA, RS, n, b - n</td>
<td>None</td>
</tr>
<tr>
<td>Insert double word from right immediate</td>
<td>inrsdi RA, RS, n, b</td>
<td>rldimi RA, RS, 64 - (b + n), b</td>
<td>None</td>
</tr>
<tr>
<td>Rotate double word left</td>
<td>rotld RA, RS, RB</td>
<td>rldcl RA, RS, RB, 0</td>
<td>None</td>
</tr>
</tbody>
</table>

**Note:** All of these extended mnemonics can be coded with a final . (period) to cause the Rc bit to be set in the underlying instruction.

**Related Information**
- Chapter 6, “Extended Instruction Mnemonics,” on page 89.
- “Extended Mnemonics of Branch Instructions” on page 89.
- “Extended Mnemonics of Condition Register Logical Instructions” on page 96.
- “Extended Mnemonics of Fixed-Point Arithmetic Instructions” on page 97.
- “Extended Mnemonics of Fixed-Point Compare Instructions” on page 98.
- “Extended Mnemonics of Fixed-Point Load Instructions” on page 99.
- “Extended Mnemonics of Fixed-Point Trap Instructions” on page 100.
- “Extended Mnemonics of Moving from or to Special-Purpose Registers” on page 102.
- “addic or ai (Add Immediate Carrying) Instruction” on page 131.
- “addic, or ai. (Add Immediate Carrying and Record) Instruction” on page 132.
- “bc (Branch Conditional) Instruction” on page 144.
- “bclr or bcr (Branch Conditional Link Register) Instruction” on page 149.
- “bcctr or bcc (Branch Conditional to Count Register) Instruction” on page 147.
- “addi or cau (Add Immediate Shifted) Instruction” on page 133.
- “cmp (Compare) Instruction” on page 156.
- “cmpeq (Compare Logical) Instruction” on page 159.
- “creqv (Condition Register Equivalent) Instruction” on page 165.
- “cror (Condition Register OR) Instruction” on page 168.
- “crnor (Condition Register NOR) Instruction” on page 167.
- “crxor (Condition Register XOR) Instruction” on page 170.
- “mfspr (Move from Special-Purpose Register) Instruction” on page 303.
- “mtspr (Move to Special-Purpose Register) Instruction” on page 315.
- “nor (NOR) Instruction” on page 333.
- “or (OR) Instruction” on page 334.
- “rlwinm or rlinm (Rotate Left Word Immediate Then AND with Mask) Instruction” on page 354.
- “tw or t (Trap Word) Instruction” on page 457.
Chapter 7. Migrating Source Programs

The assembler issues errors and warnings if a source program contains instructions that are not in the current assembly mode. Source compatibility of POWER family programs is maintained on PowerPC platforms. All POWER family user instructions are emulated in PowerPC by the operating system. Because the emulation of instructions is much slower than the execution of hardware-supported instructions, for performance reasons it may be desirable to modify the source program to use hardware-supported instructions.

The "invalid instruction form" problem occurs when restrictions are required in PowerPC but not required in POWER family. The assembler checks for invalid instruction form errors, but it cannot check the \texttt{lswx} instruction for these errors. The \texttt{lswx} instruction requires that the registers specified by the second and third operands (\texttt{RA} and \texttt{RB}) are not in the range of registers to be loaded. Since this is determined by the content of the Fixed-Point Exception Register (XER) at run time, the assembler cannot perform an invalid instruction form check for the \texttt{lswx} instruction. At run time, some of these errors may cause a silence failure, while others may cause an interruption. It may be desirable to eliminate these errors. See "Detection Error Conditions" on page 6 for more information on invalid instruction forms.

If the \texttt{mfspr} and \texttt{mtspr} instructions are used, check for proper coding of the special-purpose register (SPR) operand. The assembler requires that the low-order five bits and the high-order five bits of the SPR operand be reversed before they are used as the input operand. POWER family and PowerPC have different sets of SPR operands for nonprivileged instructions. Check for the proper encoding of these operands. Five POWER family SPRs (TID, SDR0, MQ, RTCU, and RTCL) are dropped from PowerPC, but the MQ, RTCU, and RTCL instructions are emulated in PowerPC. While these instructions can still be used, there is some performance degradation due to the emulation. (You can sometimes use the \texttt{read_real_time} and \texttt{time_base_to_time} routines instead of code accessing the real time clock or time base SPRs.)

More information on migrating source programs can be found in the following:

- "Functional Differences for POWER family and PowerPC Instructions" on page 114
- "Differences between POWER family and PowerPC Instructions with the Same Op Code" on page 115
- "Extended Mnemonics Changes" on page 116
- "POWER family Instructions Deleted from PowerPC" on page 119
- "Added PowerPC Instructions" on page 120
- "Instructions Available Only for the PowerPC 601 RISC Microprocessor" on page 121
- "Migration of Branch Conditional Statements with No Separator after Mnemonic" on page 121

Related Information

Chapter 6, "Extended Instruction Mnemonics," on page 89.

"Functional Differences for POWER family and PowerPC Instructions" on page 114.

"Differences between POWER family and PowerPC Instructions with the Same Op Code" on page 115.

"Extended Mnemonics Changes" on page 116.

"POWER family Instructions Deleted from PowerPC" on page 119.

"Added PowerPC Instructions" on page 120.

"Instructions Available Only for the PowerPC 601 RISC Microprocessor" on page 121.
Functional Differences for POWER family and PowerPC Instructions

The following table lists the POWER family and PowerPC instructions that share the same op code on POWER family and PowerPC platforms, but differ in their functional definition. Use caution when using these instructions in **com** assembly mode.

**Table 24. POWER family and PowerPC Instructions with Functional Differences**

<table>
<thead>
<tr>
<th>POWER family</th>
<th>PowerPC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcs</td>
<td>sync</td>
<td>The <em>sync</em> instruction causes more pervasive synchronization in PowerPC than the <em>dcs</em> instruction does in POWER family.</td>
</tr>
<tr>
<td>ics</td>
<td>isync</td>
<td>The <em>isync</em> instruction causes more pervasive synchronization in PowerPC than the <em>ics</em> instruction does in POWER family.</td>
</tr>
<tr>
<td>svca</td>
<td>sc</td>
<td>In POWER family, information from MSR is saved into CTR. In PowerPC, this information is saved into SRR1. POWER family only supports one vector. POWER family allows instruction fetching to continue at any of 128 locations. POWER family saves the low-order 16 bits of the instruction in CTR. PowerPC does not save the low-order 16 bits of the instruction.</td>
</tr>
<tr>
<td>mtsri</td>
<td>mtsrin</td>
<td>POWER family uses the RA field to compute the segment register number and, in some cases, the effective address (EA) is stored. POWER family has no RA field, and the EA is not stored.</td>
</tr>
<tr>
<td>lsx</td>
<td>lswx</td>
<td>POWER family does not alter the target register RT if the string length is 0. PowerPC leaves the contents of the target register RT undefined if the string length is 0.</td>
</tr>
<tr>
<td>mfsr</td>
<td>mfsr</td>
<td>This is a nonprivileged instruction in POWER family. It is a privileged instruction in PowerPC.</td>
</tr>
<tr>
<td>mfmsr</td>
<td>mfmsr</td>
<td>This is a nonprivileged instruction in POWER family. It is a privileged instruction in PowerPC.</td>
</tr>
<tr>
<td>mfdec</td>
<td>mfdec</td>
<td>The <em>mfdec</em> instruction is nonprivileged in POWER family, but becomes a privileged instruction in PowerPC. As a result, the DEC encoding number for the <em>mfdec</em> instruction is different for POWER family and PowerPC.</td>
</tr>
<tr>
<td>mffs</td>
<td>mffs</td>
<td>POWER family sets the high-order 32 bits of the result to 0xFFFF FFFF. In PowerPC, the high-order 32 bits of the result are undefined.</td>
</tr>
</tbody>
</table>

See "Features of the AIX Assembler" on page 113 for more information on the PowerPC-specific features of the assembler.

**Related Information**

- Chapter 7, “Migrating Source Programs,” on page 113.
- "Differences between POWER family and PowerPC Instructions with the Same Op Code" on page 115.
- "Extended Mnemonics Changes" on page 116.
- "POWER family Instructions Deleted from PowerPC" on page 119.
- "Added PowerPC Instructions" on page 120.
- "Instructions Available Only for the PowerPC 601 RISC Microprocessor" on page 121.
Differences between POWER family and PowerPC Instructions with the Same Op Code

This section discusses the following:
- "Instructions with the Same Op Code, Mnemonic, and Function"
- "Instructions with the Same Op Code and Function"
- "mfdec Instructions" on page 116

Instructions with the Same Op Code, Mnemonic, and Function

The following instructions are available in POWER family and PowerPC. These instructions share the same op code and mnemonic, and have the same function in POWER family and PowerPC, but use different input operand formats.
- cmp
- cmpli
- cmpl

The assembly operand format for POWER family is:

\[ BF, \ RA, \ SI \mid RB \mid UI \]

The assembly operand format for PowerPC is:

\[ BF, \ L, \ RA, \ SI \mid RB \mid UI \]

The assembler handles these as the same instructions in POWER family and PowerPC, but with different input operand formats. The \( L \) operand is one bit. For POWER family, the assembler sets this bit to 0. For 32-bit PowerPC platforms, this bit must be set to 0, or an invalid instruction form results.

Instructions with the Same Op Code and Function

The instructions listed in the following table are available in POWER family and PowerPC. These instructions share the same op code and function, but have different mnemonics and input operand formats. The assembler still places them in the POWER family/PowerPC intersection area, because the same binary code is generated. If the \(-s\) option is used, no cross-reference is given, because it is necessary to change the source code when migrating from POWER family to PowerPC, or vice versa.

![Table 25. Instructions with Same Op Code and Function](image)

<table>
<thead>
<tr>
<th>POWER family</th>
<th>PowerPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>cal</td>
<td>addi</td>
</tr>
<tr>
<td>mtsri</td>
<td>mtsrin</td>
</tr>
<tr>
<td>svca</td>
<td>sc</td>
</tr>
<tr>
<td>cau</td>
<td>addis</td>
</tr>
</tbody>
</table>

Notes:

1. \( \text{lil} \) is an extended mnemonic of \text{cal}, and \( \text{li} \) is an extended mnemonic of \text{addi}. Since the op code, function, and input operand format are the same, the assembler provides a cross-reference for \( \text{lil} \) and \( \text{li} \).
2. \( \text{liu} \) is an extended mnemonic of \text{cau}, and \( \text{lis} \) is an extended mnemonic of \text{addis}. Since the input operand format is different, the assembler does not provide a cross-reference for \( \text{liu} \) and \( \text{lis} \).
3. The immediate value for the **cau** instruction is a 16-bit unsigned integer, while the immediate value for the **addis** instruction is a 16-bit signed integer. The assembler performs a (0, 65535) value range check for the UI field and a (-32768, 32767) value range check for the SI field.

To maintain source compatibility of the **cau** and **addis** instructions, the assembler expands the value range check to (-65536, 65535) for the **addis** instruction. The sign bit is ignored and the assembler ensures only that the immediate value fits in 16 bits. This expansion does not affect the behavior of a 32-bit implementation.

For a 64-bit implementation, if bit 32 is set, it is propagated through the upper 32 bits of the 64-bit general-purpose register (GPR). Therefore, if an immediate value within the range (32768, 65535) or (-65536, -32767) is used for the **addis** instruction in a 32-bit mode, this immediate value may not be directly ported to a 64-bit mode.

**mfdec Instructions**

Moving from the DEC (decrement) special purpose register is privileged in PowerPC, but nonprivileged in POWER family. One bit in the instruction field that specifies the register is 1 for privileged operations, but 0 for nonprivileged operations. As a result, the encoding number for the DEC SPR for the **mfdec** instruction has different values in PowerPC and POWER family. The DEC encoding number is 22 for PowerPC and 6 for POWER family. If the **mfdec** instruction is used, the assembler determines the DEC encoding based on the current assembly mode. The following list shows the assembler processing of the **mfdec** instruction for each assembly mode value:

- If the assembly mode is **pwr**, **pwr2**, or **601**, the DEC encoding is 6.
- If the assembly mode is **ppc**, **603**, or **604**, the DEC encoding is 22.
- If the default assembly mode, which treats POWER family/PowerPC incompatibility errors as instructional warnings, is used, the DEC encoding is 6. Instructional warning 158 reports that the DEC SPR encoding 6 is used to generate the object code. The warning can be suppressed with the `-W` flag.
- If the assembly mode is **any**, the DEC encoding is 6. If the `-w` flag is used, a warning message (158) reports that the DEC SPR encoding 6 is used to generate the object code.
- If the assembly mode is **com**, an error message reports that the **mfdec** instruction is not supported. No object code is generated. In this situation, the **mfspr** instruction must be used to encode the DEC number.

**Related Information**

- [Chapter 7, “Migrating Source Programs,” on page 113](#)
- [“Functional Differences for POWER family and PowerPC Instructions” on page 114](#)
- [“Extended Mnemonics Changes.”](#)
- [“POWER family Instructions Deleted from PowerPC” on page 119](#)
- [“Added PowerPC Instructions” on page 120](#)
- [“Instructions Available Only for the PowerPC 601 RISC Microprocessor” on page 121](#)

**Extended Mnemonics Changes**

The following lists show the added extended mnemonics for POWER family and PowerPC. The assembler places all POWER family and PowerPC extended mnemonics in the POWER family/PowerPC intersection area if their basic mnemonics are in this area. Extended mnemonics are separated for POWER family and PowerPC only for migration purposes. See [Chapter 6, “Extended Instruction Mnemonics,” on page 89](#) for more information.
Extended Mnemonics in com Mode
The following PowerPC extended mnemonics for branch conditional instructions have been added:

- bdzt
- bdztla
- bdztl
- bdztla
- bdzf
- bdzfa
- bdzf
- bdzf
- bdzfl
- bdzfla
- bdztl
- bdznza
- bdnztl
- bdnztl
- bdznzfl
- bdznzfla
- bdztlr
- bdztlrl
- bdzflr
- bdzflrl
- bdznzflr
- bdznzflrl
- bun
- buna
- bunl
- bunla
- bunlr
- bunlrl
- buncr
- buncrl
- bnu
- bnu
- bnul
- bnula
- bnulr
- bnulrl
- bnuctr
- bnuctrl
- crset

The following PowerPC extended mnemonics for condition register logical instructions have been added:
The following PowerPC extended mnemonics for fixed-point load instructions have been added:
- `li`
- `lis`
- `la`

The following PowerPC extended mnemonics for fixed-point arithmetic instructions have been added:
- `subi`
- `subis`
- `subc`

The following PowerPC extended mnemonics for fixed-point compare instructions have been added:
- `cmpwi`
- `cmpw`
- `cmplwi`
- `cmplw`

The following PowerPC extended mnemonics for fixed-point trap instructions have been added:
- `trap`
- `twling`
- `twlingi`
- `twlinl`
- `twlinli`
- `twng`
- `twngi`
- `twnl`
- `twnli`

The following PowerPC extended mnemonics for fixed-point logical instructions have been added:
- `nop`
- `mr[.]`
- `not[.]`

The following PowerPC extended mnemonics for fixed-point rotate and shift instructions have been added:
- `extlwi[.]`
- `extrwi[.]`
- `inslwi[.]`
- `insrwi[.]`
- `rotlwi[.]`
- `rothwi[.]`
- `rotlwi[.]`
- `clrlwi[.]`
- `clrrwi[.]`
- `clrlsiwi[.]`
Extended Mnemonics in ppc Mode
The following PowerPC extended mnemonic for fixed-point arithmetic instructions has been added for ppc mode:

- sub

Related Information
Chapter 7, “Migrating Source Programs,” on page 113.
“Functional Differences for POWER family and PowerPC Instructions” on page 114.
“Differences between POWER family and PowerPC Instructions with the Same Op Code” on page 115.
“POWER family Instructions Deleted from PowerPC.”
“Added PowerPC Instructions” on page 120.
“Instructions Available Only for the PowerPC 601 RISC Microprocessor” on page 121.
Chapter 6, “Extended Instruction Mnemonics,” on page 89.

POWER family Instructions Deleted from PowerPC
The following table lists the POWER family instructions that have been deleted from PowerPC, yet are still supported by the PowerPC 601 RISC Microprocessor. AIX provides services to emulate most of these instructions if an attempt to execute one of them is made on a processor that does not include the instruction, such as PowerPC 603 RISC Microprocessor or PowerPC 604 RISC Microprocessor, but no emulation services are provided for the mtrtcl, mtrtcu, or svcla instructions. Using the code to emulate an instruction is much slower than executing an instruction.

Table 26. POWER family Instructions Deleted from PowerPC, Supported by PowerPC 601 RISC Microprocessor

<table>
<thead>
<tr>
<th>abs[o][.]</th>
<th>clcs</th>
<th>div[o][.]</th>
<th>divs[o][.]</th>
</tr>
</thead>
<tbody>
<tr>
<td>doz[o][.]</td>
<td>dozi</td>
<td>lscbx[.]</td>
<td>maskg[.]</td>
</tr>
<tr>
<td>maskir[.]</td>
<td>mfmq</td>
<td>mfrtcl</td>
<td>mfrtcu</td>
</tr>
<tr>
<td>mtmq</td>
<td>mtrtcl</td>
<td>mtrtcu</td>
<td>mul[o][.]</td>
</tr>
<tr>
<td>nabs[o][.]</td>
<td>rmi[.]</td>
<td>rrrib[.]</td>
<td>sle[.]</td>
</tr>
<tr>
<td>sleq[.]</td>
<td>sliq[.]</td>
<td>sliq[.]</td>
<td>sliq[.]</td>
</tr>
<tr>
<td>s1q[.]</td>
<td>sraiq[.]</td>
<td>sraq[.]</td>
<td>sre[.]</td>
</tr>
<tr>
<td>srea[.]</td>
<td>sreq[.]</td>
<td>sriq[.]</td>
<td>srlq[.]</td>
</tr>
<tr>
<td>srlq[.]</td>
<td>sq[.]</td>
<td>svcla</td>
<td></td>
</tr>
</tbody>
</table>

Note: Extended mnemonics are not included in the previous table, except for extended mnemonics for the mfspr and mtspr instructions.

The following table lists the POWER family instructions that have been deleted from PowerPC and that are not supported by the PowerPC 601 RISC Microprocessor. AIX does not provide services to emulate most of these instructions. However, emulation services are provided for the clf, dclst, and dclz instructions. Also, the cli instruction is emulated, but only when it is executed in privileged mode.

Table 27. POWER family Instructions Deleted from PowerPC, Not Supported by PowerPC 601 RISC Microprocessor

<table>
<thead>
<tr>
<th>clf</th>
<th>cli</th>
<th>dclst</th>
<th>dclz</th>
</tr>
</thead>
<tbody>
<tr>
<td>mfsdr0</td>
<td>mfsri</td>
<td>mftid</td>
<td>mtsdr0</td>
</tr>
</tbody>
</table>
Table 27. POWER family Instructions Deleted from PowerPC, Not Supported by PowerPC 601 RISC Microprocessor (continued)

<table>
<thead>
<tr>
<th>mttid</th>
<th>rac[.]</th>
<th>rfsvc</th>
<th>svc</th>
</tr>
</thead>
<tbody>
<tr>
<td>svcl</td>
<td>tlb</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Related Information

[Chapter 7, “Migrating Source Programs,” on page 113.]

“Functional Differences for POWER family and PowerPC Instructions” on page 114.

“Differences between POWER family and PowerPC Instructions with the Same Op Code” on page 115.

“Extended Mnemonics Changes” on page 116.

“Added PowerPC Instructions.”

“Instructions Available Only for the PowerPC 601 RISC Microprocessor” on page 121.

Added PowerPC Instructions

The following table lists instructions that have been added to PowerPC, but are not in POWER family. These instructions are supported by the PowerPC 601 RISC Microprocessor.

Table 28. Added PowerPC Instructions, Supported by PowerPC 601 RISC Microprocessor

<table>
<thead>
<tr>
<th>dcbf</th>
<th>dcbi</th>
<th>dcbst</th>
<th>dcbt</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcbst</td>
<td>dcbz</td>
<td>divw[0][.]</td>
<td>divwu[0][.]</td>
</tr>
<tr>
<td>eieio</td>
<td>extsb[.]</td>
<td>fadds[.]</td>
<td>fdivs[.]</td>
</tr>
<tr>
<td>fmadds[.]</td>
<td>fmsubs[.]</td>
<td>fmuls[.]</td>
<td>fnmadds[.]</td>
</tr>
<tr>
<td>fnmsubs[.]</td>
<td>fsubs[.]</td>
<td>icbi</td>
<td>lwax</td>
</tr>
<tr>
<td>mfear</td>
<td>mfpvr</td>
<td>mfsprg</td>
<td>mfsrin</td>
</tr>
<tr>
<td>mtear</td>
<td>mtsprg</td>
<td>mulh[0][.]</td>
<td>mulhw[0][.]</td>
</tr>
<tr>
<td>stwcx.</td>
<td>subf[0][.]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Extended mnemonics are not included in the previous table, except for extended mnemonics for the mfspr and mtspr instructions.

The following table lists instructions that have been added to PowerPC, but are not in POWER family. These instructions are not supported by the PowerPC 601 RISC Microprocessor.

Table 29. PowerPC Instructions, Not Supported by PowerPC 601 RISC Microprocessor

<table>
<thead>
<tr>
<th>mfdbatl</th>
<th>mfdbatu</th>
<th>mtdbatl</th>
<th>mtdbatu</th>
</tr>
</thead>
<tbody>
<tr>
<td>mttb</td>
<td>mttbu</td>
<td>mttb</td>
<td>mttbu</td>
</tr>
<tr>
<td>mfibatl</td>
<td>mfibatu</td>
<td>mtibatl</td>
<td>mtibatu</td>
</tr>
</tbody>
</table>

Related Information

[Chapter 7, “Migrating Source Programs,” on page 113.]

“Functional Differences for POWER family and PowerPC Instructions” on page 114.
Instructions Available Only for the PowerPC 601 RISC Microprocessor

The following table lists PowerPC optional instructions that are implemented in the PowerPC 601 RISC Microprocessor:

<table>
<thead>
<tr>
<th>ecixw</th>
<th>ecowx</th>
<th>mfbatl</th>
<th>mfbatu</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtbatl</td>
<td>mtbatu</td>
<td>tlbie</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Extended mnemonics, with the exception of mfspr and mtspr extended mnemonics, are not provided.

Related Information

Chapter 7, “Migrating Source Programs,” on page 113.

Migration of Branch Conditional Statements with No Separator after Mnemonic

The AIX assembler may parse some statements different from the previous version of the assembler. This different parsing is only a possibility for statements that meet all the following conditions:

- The statement does not have a separator character (space or tab) between the mnemonic and the operands.
- The first character of the first operand is a plus sign (+) or a minus sign (-).
- The mnemonic represents a Branch Conditional instruction.

If an assembler program has statements that meet all the conditions above, and the minus sign, or a plus sign in the same location, is intended to be part of the operands, not part of the mnemonic, the source program must be modified. This is especially important for minus signs, because moving a minus sign can significantly change the meaning of a statement.

The possibility of different parsing occurs in AIX because the assembler was modified to support branch prediction extended mnemonics which use the plus sign and minus sign as part of the mnemonic. In previous versions of the assembler, letters and period (.) were the only possible characters in mnemonics.

For information, see "Extended Mnemonics for Branch Prediction" on page 93.
Examples

1. The following statement is parsed by the AIX assembler so that the minus sign is part of the mnemonic (but previous versions of the assembler parsed the minus sign as part of the operands) and must be modified if the minus sign is intended to be part of the operands:

```assembly
bnea- 16 # Separator after the -, but none before
   # Now: bnea- is a Branch Prediction Mnemonic
   # and 16 is operand.
   # Previously: bnea was mnemonic
   # and -16 was operand.
```

2. The following are several sample statements which the AIX assembler parses the same as previous assemblers (the minus sign will be interpreted as part of the operands):

```assembly
bnea -16 # Separator in source program - Good practice
bnea-16   # No separators before or after minus sign
bnea - 16 # Separators before and after the minus sign
```

Related Information

“Features of the AIX Assembler” on page 1.

“Extended Mnemonics for Branch Prediction” on page 93.
Chapter 8. Instruction Set

This chapter contains reference articles for the operating system assembler instruction set. The following appendixes also provide information on the operating system assembler instruction set:

- Appendix B. Instruction Set Sorted by Mnemonic
- Appendix C. Instruction Set Sorted by Primary and Extended Op Code
- Appendix D. Instructions Common to POWER family, POWER2, and PowerPC
- Appendix E. POWER family and POWER2 Instructions
- Appendix F. PowerPC Instructions
- Appendix G. PowerPC 601 RISC Microprocessor Instructions
- Appendix I, “Vector Processor,” on page 597

abs (Absolute) Instruction

Purpose
Takes the absolute value of the contents of a general-purpose register and places the result in another general-purpose register.

Note: The abs instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>360</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family

<table>
<thead>
<tr>
<th>abs</th>
<th>RT</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs.</td>
<td>RT</td>
<td>RA</td>
</tr>
<tr>
<td>abso</td>
<td>RT</td>
<td>RA</td>
</tr>
<tr>
<td>abso.</td>
<td>RT</td>
<td>RA</td>
</tr>
</tbody>
</table>

Description

The abs instruction places the absolute value of the contents of general-purpose register (GPR) RA into the target GPR RT.

If GPR RA contains the most negative number (‘8000 0000’), the result of the instruction is the most negative number, and the instruction will set the Overflow bit in the Fixed-Point Exception Register to 1 if the OE bit is set to 1.

The abs instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.
The four syntax forms of the **abs** instruction always affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

### Parameters

**RT**  
Specifies the target general-purpose register where result of operation is stored.

**RA**  
Specifies the source general-purpose register for operation.

### Examples

1. The following code takes the absolute value of the contents of GPR 4 and stores the result in GPR 6:

   ```
   # Assume GPR 4 contains 0x7000 3000.
   abs 6,4
   # GPR 6 now contains 0x7000 3000.
   ```

2. The following code takes the absolute value of the contents of GPR 4, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0xFFFF FFFF.
   abs. 6,4
   # GPR 6 now contains 0x0000 0001.
   ```

3. The following code takes the absolute value of the contents of GPR 4, stores the result in GPR 6, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x8004 3000.
   abso 6,4
   # GPR 6 now contains 0x4FFB D000.
   ```

4. The following code takes the absolute value of the contents of GPR 4, stores the result in GPR 6, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x8000 0000.
   abso. 6,4
   # GPR 6 now contains 0x8000 0000.
   ```

### Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Arithmetic Instructions](#)

---

**add (Add) or cax (Compute Address) Instruction**

### Purpose

Adds the contents of two general-purpose registers.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>266</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC

\[
\text{add} \quad RT, RA, RB \\
\text{add.} \quad RT, RA, RB \\
\text{addo} \quad RT, RA, RB \\
\text{addo.} \quad RT, RA, RB
\]

POWER family

\[
\text{cax} \quad RT, RA, RB \\
\text{cax.} \quad RT, RA, RB \\
\text{caxo} \quad RT, RA, RB \\
\text{caxo.} \quad RT, RA, RB
\]

Description

The **add** and **cax** instructions place the sum of the contents of general-purpose register (GPR) RA and GPR RB into the target GPR RT.

The **add** and **cax** instructions have four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>0</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>add.</td>
<td>0</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>addo</td>
<td>1</td>
<td>SO,OV</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>addo.</td>
<td>1</td>
<td>SO,OV</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>cax</td>
<td>0</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>cax.</td>
<td>0</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>caxo</td>
<td>1</td>
<td>SO,OV</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>caxo.</td>
<td>1</td>
<td>SO,OV</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The four syntax forms of the **add** instruction and the four syntax forms of the **cax** instruction never affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.
Parameters

**RT** Specifies target general-purpose register where result of operation is stored.

**RA** Specifies source general-purpose register for operation.

**RB** Specifies source general-purpose register for operation.

Examples

1. The following code adds the address or contents in GPR 6 to the address or contents in GPR 3 and stores the result in GPR 4:
   
   ```
   # Assume GPR 6 contains 0x0004 0000.
   # Assume GPR 3 contains 0x0000 4000.
   add 4,6,3
   # GPR 4 now contains 0x0004 4000.
   ```

2. The following code adds the address or contents in GPR 6 to the address or contents in GPR 3, stores the result in GPR 4, and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 6 contains 0x8000 7000.
   # Assume GPR 3 contains 0x7000 8000.
   add 4,6,3
   # GPR 4 now contains 0xF000 F000.
   ```

3. The following code adds the address or contents in GPR 6 to the address or contents in GPR 3, stores the result in GPR 4, and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register to reflect the result of the operation:

   ```
   # Assume GPR 6 contains 0xEFFF FFFF.
   # Assume GPR 3 contains 0x8000 0000.
   addo 4,6,3
   # GPR 4 now contains 0x6FFF FFFF.
   ```

4. The following code adds the address or contents in GPR 6 to the address or contents in GPR 3, stores the result in GPR 4, and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 6 contains 0xEFFF FFFF.
   # Assume GPR 3 contains 0xEFFF FFFF.
   addo 4,6,3
   # GPR 4 now contains 0xDFFF FFFE.
   ```

Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Address Computation Instructions](#)

**addc or a (Add Carrying) Instruction**

**Purpose**

Adds the contents of two general-purpose registers and places the result in a general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
</tbody>
</table>
### Description

The `addc` and `a` instructions place the sum of the contents of general-purpose register (GPR) `RA` and GPR `RB` into the target GPR `RT`.

The `addc` instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

The `a` instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addc</code></td>
<td>0</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>addc</code></td>
<td>0</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td><code>addco</code></td>
<td>1</td>
<td>SO,OV,CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>addco</code></td>
<td>1</td>
<td>SO,OV,CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td><code>a</code></td>
<td>0</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>a</code></td>
<td>0</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td><code>ao</code></td>
<td>1</td>
<td>SO,OV,CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>ao</code></td>
<td>1</td>
<td>SO,OV,CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The four syntax forms of the `addc` instruction and the four syntax forms of the `a` instruction always affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

### Parameters

- **RT** Specifies target general-purpose register where result of operation is stored.
- **RA** Specifies source general-purpose register for operation.
RB

Specifies source general-purpose register for operation.

Examples
1. The following code adds the contents of GPR 4 to the contents of GPR 10 and stores the result in GPR 6:

   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 10 contains 0x8000 7000.
   addc 6,4,10
   # GPR 6 now contains 0x1000 A000.

2. The following code adds the contents of GPR 4 to the contents of GPR 10, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   # Assume GPR 4 contains 0x7000 3000.
   # Assume GPR 10 contains 0xFFFF FFFF.
   addc. 6,4,10
   # GPR 6 now contains 0x7000 2FFF.

3. The following code adds the contents of GPR 4 to the contents of GPR 10, stores the result in GPR 6, and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register to reflect the result of the operation:

   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 10 contains 0x7B41 92C0.
   addco 6,4,10
   # GPR 6 now contains 0x0B41 C2C0.

4. The following code adds the contents of GPR 4 to the contents of GPR 10, stores the result in GPR 6, and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

   # Assume GPR 4 contains 0x8000 0000.
   # Assume GPR 10 contains 0x8000 7000.
   addco. 6,4,10
   # GPR 6 now contains 0x0000 7000.

Related Information

[Fixed-Point Processor](#)

[Fixed-Point Arithmetic Instructions](#)

adde or ae (Add Extended) Instruction

Purpose

Adds the contents of two general-purpose registers to the value of the Carry bit in the Fixed-Point Exception Register and places the result in a general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>138</td>
</tr>
<tr>
<td>Bits</td>
<td>Value</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**PowerPC**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addde</td>
<td>RT RA RB</td>
<td>Adds the contents of RA and RB to RT and sets Carry bit (CA).</td>
</tr>
<tr>
<td>add.</td>
<td>RT RA RB</td>
<td>Adds the contents of RA and RB to RT and sets Carry bit (CA). If OE is 1, sets Summary Overflow (SO) and Overflow (OV) bits.</td>
</tr>
<tr>
<td>adddeo</td>
<td>RT RA RB</td>
<td>Adds the contents of RA and RB to RT and sets Carry bit (CA). If OE is 1, sets Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits.</td>
</tr>
<tr>
<td>addeo.</td>
<td>RT RA RB</td>
<td>Adds the contents of RA and RB to RT and sets Carry bit (CA). If OE is 1, sets Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits.</td>
</tr>
</tbody>
</table>

**POWER family**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ae</td>
<td>RT RA RB</td>
<td>Adds the contents of RA and RB to RT and sets Carry bit (CA).</td>
</tr>
<tr>
<td>ae.</td>
<td>RT RA RB</td>
<td>Adds the contents of RA and RB to RT and sets Carry bit (CA). If OE is 1, sets Summary Overflow (SO) and Overflow (OV) bits.</td>
</tr>
<tr>
<td>aeo</td>
<td>RT RA RB</td>
<td>Adds the contents of RA and RB to RT and sets Carry bit (CA). If OE is 1, sets Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits.</td>
</tr>
<tr>
<td>aeo.</td>
<td>RT RA RB</td>
<td>Adds the contents of RA and RB to RT and sets Carry bit (CA). If OE is 1, sets Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits.</td>
</tr>
</tbody>
</table>

**Description**

The **addde** and **ae** instructions place the sum of the contents of general-purpose register (GPR) RA, GPR RB, and the Carry bit into the target GPR RT.

The **addde** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

The **ae** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>addde</td>
<td>0</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>add.</td>
<td>0</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>adddeo</td>
<td>1</td>
<td>SO,OV,CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>addeo.</td>
<td>1</td>
<td>SO,OV,CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>ae</td>
<td>0</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>ae.</td>
<td>0</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>aeo</td>
<td>1</td>
<td>SO,OV,CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>aeo.</td>
<td>1</td>
<td>SO,OV,CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The four syntax forms of the **addde** instruction and the four syntax forms of the **ae** instruction always affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- **RT** Specifies target general-purpose register where result of operation is stored.
- **RA** Specifies source general-purpose register for operation.
- **RB** Specifies source general-purpose register for operation.
Examples
1. The following code adds the contents of GPR 4, the contents of GPR 10, and the Fixed-Point Exception Register Carry bit and stores the result in GPR 6:

```assembly
# Assume GPR 4 contains 0x1000 0400.
# Assume GPR 10 contains 0x1000 0400.
# Assume the Carry bit is one.
adde 6,4,10  # GPR 6 now contains 0x2000 0801.
```

2. The following code adds the contents of GPR 4, the contents of GPR 10, and the Fixed-Point Exception Register Carry bit; stores the result in GPR 6; and sets Condition Register Field 0 to reflect the result of the operation:

```assembly
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 10 contains 0x7B41 92C0.
# Assume the Carry bit is zero.
adde 6,4,10  # GPR 6 now contains 0x0B41 C2C0.
```

3. The following code adds the contents of GPR 4, the contents of GPR 10, and the Fixed-Point Exception Register Carry bit; stores the result in GPR 6; and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register to reflect the result of the operation:

```assembly
# Assume GPR 4 contains 0x1000 0400.
# Assume GPR 10 contains 0xEFFF FFFF.
# Assume the Carry bit is one.
addeo 6,4,10  # GPR 6 now contains 0x0000 0400.
```

4. The following code adds the contents of GPR 4, the contents of GPR 10, and the Fixed-Point Exception Register Carry bit; stores the result in GPR 6; and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```assembly
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 10 contains 0x8000 7000.
# Assume the Carry bit is zero.
addeo 6,4,10  # GPR 6 now contains 0x1000 A000.
```

Related Information
- [Fixed-Point Processor](#)
- [Fixed-Point Arithmetic Instructions](#)

### addi (Add Immediate) or cal (Compute Address Lower) Instruction

**Purpose**
Calculates an address from an offset and a base address and places the result in a general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>14</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>SI/D</td>
</tr>
</tbody>
</table>
PowerPC
addi  \( RT \, RA \, SI \)

POWER family
cal  \( RT \, D \, RA \)

See [Extended Mnemonics of Fixed-Point Arithmetic Instructions](#) and [Extended Mnemonics of Fixed-Point Load Instructions](#) for more information.

**Description**

The `addi` and `cal` instructions place the sum of the contents of general-purpose register (GPR) `RA` and the 16-bit two's complement integer `SI` or `D`, sign-extended to 32 bits, into the target GPR `RT`. If GPR `RA` is GPR 0, then `SI` or `D` is stored into the target GPR `RT`.

The `addi` and `cal` instructions have one syntax form and do not affect Condition Register Field 0 or the Fixed-Point Exception Register.

**Parameters**

- **RT**: Specifies target general-purpose register where result of operation is stored.
- **RA**: Specifies source general-purpose register for operation.
- **D**: Specifies 16-bit two’s complement integer sign extended to 32 bits.
- **SI**: Specifies 16-bit signed integer for operation.

**Examples**

The following code calculates an address or contents with an offset of 0xFFFF 8FF0 from the contents of GPR 5 and stores the result in GPR 4:

```plaintext
# Assume GPR 5 contains 0x0000 0900.
addi 4,0xFFFF8FF0(5)
# GPR 4 now contains 0xFFFF 98F0.
```

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Address Computation Instructions](#)

---

### addic or ai (Add Immediate Carrying) Instruction

**Purpose**

Adds the contents of a general-purpose register and a 16-bit signed integer, places the result in a general-purpose register, and effects the Carry bit of the Fixed-Point Exception Register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>12</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>SI</td>
</tr>
</tbody>
</table>
The `addic` and `ai` instructions place the sum of the contents of general-purpose register (GPR) `RA` and a 16-bit signed integer, `SI`, into target GPR `RT`.

The 16-bit integer provided as immediate data is sign-extended to 32 bits prior to carrying out the addition operation.

The `addic` and `ai` instructions have one syntax form and can set the Carry bit of the Fixed-Point Exception Register; these instructions never affect Condition Register Field 0.

### Parameters

- **RT**  Specifies target general-purpose register where result of operation is stored.
- **RA**  Specifies source general-purpose register for operation.
- **SI**  Specifies 16-bit signed integer for operation.

### Examples

The following code adds 0xFFFF FFFF to the contents of GPR 4, stores the result in GPR 6, and sets the Carry bit to reflect the result of the operation:

```assembly
# Assume GPR 4 contains 0x0000 2346.
addic 6,4,0xFFFFFFFF
# GPR 6 now contains 0x0000 2345.
```

### Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Arithmetic Instructions](#)

---

### `addic`. or `ai`. (Add Immediate Carrying and Record) Instruction

#### Purpose

Performs an addition with carry of the contents of a general-purpose register and an immediate value.

#### Syntax

<table>
<thead>
<tr>
<th></th>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
<td></td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
<td></td>
</tr>
<tr>
<td>16-31</td>
<td>SI</td>
<td></td>
</tr>
</tbody>
</table>
The `addic.` and `ai.` instructions place the sum of the contents of general-purpose register (GPR) `RA` and a 16-bit signed integer, `SI`, into the target GPR `RT`.

The 16-bit integer `SI` provided as immediate data is sign-extended to 32 bits prior to carrying out the addition operation.

The `addic.` and `ai.` instructions have one syntax form and can set the Carry Bit of the Fixed-Point Exception Register. These instructions also affect Condition Register Field 0.

Parameters

- `RT` Specifies target general-purpose register where result of operation is stored.
- `RA` Specifies source general-purpose register for operation.
- `SI` Specifies 16-bit signed integer for operation.

Examples

The following code adds a 16-bit signed integer to the contents of GPR 4, stores the result in GPR 6, and sets the Fixed-Point Exception Register Carry bit and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xEFFF FFFF.
addic. 6,4,0x1000
# GPR 6 now contains 0xF000 0FFF.
```

Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Arithmetic Instructions](#)

### addis or cau (Add Immediate Shifted) Instruction

#### Purpose

Calculates an address from a concatenated offset and a base address and loads the result in a general-purpose register.

#### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>15</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>Bits</td>
<td>Value</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>16-31</td>
<td>SI/UI</td>
</tr>
</tbody>
</table>

PowerPC

**addis**  
\[ RT | RA | S \]

**POWER family**

**cau**  
\[ RT | RA | UI \]

See [Extended Mnemonics of Fixed-Point Arithmetic Instructions](#) and [Extended Mnemonics of Fixed-Point Load Instructions](#) for more information.

**Description**

The **addis** and **cau** instructions place the sum of the contents of general-purpose register (GPR) \( RA \) and the concatenation of a 16-bit unsigned integer, \( SI \) or \( UI \), and \( x'0000' \) into the target GPR \( RT \). If GPR \( RA \) is GPR 0, then the sum of the concatenation of 0, \( SI \) or \( UI \), and \( x'0000' \) is stored into the target GPR \( RT \).

The **addis** and **cau** instructions have one syntax form and do not affect Condition Register Field 0 or the Fixed-Point Exception Register.

**Note:** The immediate value for the **cau** instruction is a 16-bit unsigned integer, whereas the immediate value for the **addis** instruction is a 16-bit signed integer. This difference is a result of extending the architecture to 64 bits.

The assembler does a 0 to 65535 value-range check for the \( UI \) field, and a -32768 to 32767 value-range check for the \( SI \) field.

To keep the source compatibility of the **addis** and **cau** instructions, the assembler expands the value-range check for the **addis** instruction to -65536 to 65535. The sign bit is ignored and the assembler only ensures that the immediate value fits into 16 bits. This expansion does not affect the behavior of a 32-bit implementation or 32-bit mode in a 64-bit implementation.

The **addis** instruction has different semantics in 32-bit mode than it does in 64-bit mode. If bit 32 is set, it propagates through the upper 32 bits of the 64-bit general-purpose register. Use caution when using the **addis** instruction to construct an unsigned integer. The **addis** instruction with an unsigned integer in 32-bit may not be directly ported to 64-bit mode. The code sequence needed to construct an unsigned integer in 64-bit mode is significantly different from that needed in 32-bit mode.

**Parameters**

- **\( RT \)** Specifies target general-purpose register where result of operation is stored.
- **\( RA \)** Specifies first source general-purpose register for operation.
- **\( UI \)** Specifies 16-bit unsigned integer for operation.
- **\( SI \)** Specifies 16-bit signed integer for operation.

**Examples**

The following code adds an offset of 0x0011 0000 to the address or contents contained in GPR 6 and loads the result into GPR 7:

\[
# \text{Assume GPR 6 contains 0x0000 4000.} \\
\text{addis 7,6,0x0011} \\
# GPR 7 now contains 0x0011 4000. 
\]
addme or ame (Add to Minus One Extended) Instruction

Purpose
Adds the contents of a general-purpose register, the Carry bit in the Fixed-Point Exception Register, and -1 and places the result in a general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>234</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC
addme  \( RT \), RA
addme. \( RT \), RA
addmeO  \( RT \), RA
addmeO. \( RT \), RA

POWER family
ame  \( RT \), RA
ame. \( RT \), RA
ameO  \( RT \), RA
ameO. \( RT \), RA

Description
The addme and ame instructions place the sum of the contents of general-purpose register (GPR) RA, the Carry bit of the Fixed-Point Exception Register, and -1 (0xFFFF FFFF) into the target GPR RT.

The addme instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

The ame instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>addme</td>
<td>0</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>addme.</td>
<td>0</td>
<td>CA</td>
<td>1</td>
<td>LT, GT, EQ, SO</td>
</tr>
</tbody>
</table>
The four syntax forms of the `addme` instruction and the four syntax forms of the `ame` instruction always affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- **RT** Specifies target general-purpose register where result of operation is stored.
- **RA** Specifies source general-purpose register for operation.

**Examples**

1. The following code adds the contents of GPR 4, the Carry bit in the Fixed-Point Exception Register, and -1 and stores the result in GPR 6:
   ```
   # Assume GPR 4 contains 0x9000 3000.
   # Assume the Carry bit is zero.
   addme 6,4
   # GPR 6 now contains 0x9000 2FFF.
   ```

2. The following code adds the contents of GPR 4, the Carry bit in the Fixed-Point Exception Register, and -1; stores the result in GPR 6; and sets Condition Register Field 0 to reflect the result of the operation:
   ```
   # Assume GPR 4 contains 0xB000 42FF.
   # Assume the Carry bit is zero.
   addme. 6,4
   # GPR 6 now contains 0xB000 42FE.
   ```

3. The following code adds the contents of GPR 4, the Carry bit in the Fixed-Point Exception Register, and -1; stores the result in GPR 6; and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register to reflect the result of the operation:
   ```
   # Assume GPR 4 contains 0x8000 0000.
   # Assume the Carry bit is zero.
   ameo 6,4
   # GPR 6 now contains 0x7FFF FFFF.
   ```

4. The following code adds the contents of GPR 4, the Carry bit in the Fixed-Point Exception Register, and -1; stores the result in GPR 6; and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:
   ```
   # Assume GPR 4 contains 0x8000 0000.
   # Assume the Carry bit is one.
   ameo. 6,4
   # GPR 6 now contains 0x8000 000.
   ```

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Arithmetic Instructions](#)
addze or aze (Add to Zero Extended) Instruction

Purpose
Adds the contents of a general-purpose register, zero, and the value of the Carry bit in the Fixed-Point Exception Register and places the result in a general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>202</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC
addze
addze.
addzeo
addzeo.

POWER family
aze
aze.
azeo
azeo.

Description
The addze and aze instructions add the contents of general-purpose register (GPR) RA, the Carry bit, and 0x0000 0000 and place the result into the target GPR RT.

The addze instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

The aze instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>addze</td>
<td>0</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>addze.</td>
<td>0</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>addzeo</td>
<td>1</td>
<td>SO,OV,CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>addzeo.</td>
<td>1</td>
<td>SO,OV,CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>aze</td>
<td>0</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>aze.</td>
<td>0</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>
The four syntax forms of the **addze** instruction and the four syntax forms of the **aze** instruction always affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

### Parameters

**RT**  
Specifies target general-purpose register where result of operation is stored.

**RA**  
Specifies source general-purpose register for operation.

### Examples

1. The following code adds the contents of GPR 4, 0, and the Carry bit and stores the result in GPR 6:

   ```
   # Assume GPR 4 contains 0x7B41 92C0.
   # Assume the Carry bit is zero.
   addz 6,4  
   # GPR 6 now contains 0x7B41 92C0.
   ```

2. The following code adds the contents of GPR 4, 0, and the Carry bit, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0xEFFF FFFF.
   # Assume the Carry bit is one.
   addz 6,4  
   # GPR 6 now contains 0xF000 0000.
   ```

3. The following code adds the contents of GPR 4, 0, and the Carry bit; stores the result in GPR 6; and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x9000 3000.
   # Assume the Carry bit is one.
   addzeo 6,4  
   # GPR 6 now contains 0x9000 3001.
   ```

4. The following code adds the contents of GPR 4, 0, and the Carry bit; stores the result in GPR 6; and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0xEFFF FFFF.
   # Assume the Carry bit is zero.
   adzeo 6,4  
   # GPR 6 now contains 0xEFFF FFFF.
   ```

### Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Arithmetic Instructions](#)

### and (AND) Instruction

#### Purpose

Logically ANDs the contents of two general-purpose registers and places the result in a general-purpose register.
### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>28</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

The instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>and.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

### Description

The **and** instruction logically ANDs the contents of general-purpose register (GPR) RS with the contents of GPR RB and places the result into the target GPR RA.

The **and** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

### Parameters

- **RA** Specifies target general-purpose register where result of operation is stored.
- **RS** Specifies source general-purpose register for operation.
- **RB** Specifies source general-purpose register for operation.

### Examples

1. The following code logically ANDs the contents of GPR 4 with the contents of GPR 7 and stores the result in GPR 6:

```plaintext
# Assume GPR 4 contains 0xFFF2 5730.
# Assume GPR 7 contains 0x7B41 92C0.
and 6,4,7
# GPR 6 now contains 0x7B40 1200.
```

2. The following code logically ANDs the contents of GPR 4 with the contents of GPR 7, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

```plaintext
# Assume GPR 4 contains 0xFFF2 5730.
# Assume GPR 7 contains 0xFFFF EFFF.
and. 6,4,7
# GPR 6 now contains 0xFFF2 4730.
```
Related Information

- Fixed-Point Processor
- Fixed-Point Logical Instructions

**andc (AND with Complement) Instruction**

**Purpose**
Logically ANDs the contents of a general-purpose register with the complement of the contents of a general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>60</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

```
andc  RA RS RB
andc. RA RS RB
```

**Description**
The `andc` instruction logically ANDs the contents of general-purpose register (GPR) `RS` with the complement of the contents of GPR `RB` and places the result into GPR `RA`.

The `andc` instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>andc</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>andc.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the `andc` instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- `RA` Specifies target general-purpose register where result of operation is stored.
- `RS` Specifies source general-purpose register for operation.
- `RB` Specifies source general-purpose register for operation.
Examples

1. The following code logically ANDs the contents of GPR 4 with the complement of the contents of GPR 5 and stores the result in GPR 6:

   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 5 contains 0xFFFF FFFF.
   # The complement of 0xFFFF FFFF becomes 0x0000 0000.
   andc 6,4,5
   # GPR 6 now contains 0x0000 0000.

2. The following code logically ANDs the contents of GPR 4 with the complement of the contents of GPR 5, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   # Assume GPR 4 contains 0xB004 3000.
   # Assume GPR 5 contains 0x7676 7676.
   # The complement of 0x7676 7676 is 0x8989 8989.
   andc 6,4,5
   # GPR 6 now contains 0x8000 0000.

Related Information

Fixed-Point Processor.

Fixed-Point Logical Instructions.

andi. or andil. (AND Immediate) Instruction

Purpose

Logically ANDs the contents of a general-purpose register with an immediate value.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>28</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>UI</td>
</tr>
</tbody>
</table>

PowerPC

andi.  

andil.  

POWER family

andi.  

andil.  

Description

The andi. and andil. instructions logically AND the contents of general-purpose register (GPR) RS with the concatenation of x’0000’ and a 16-bit unsigned integer, UI, and place the result in GPR RA.

The andi. and andil. instructions have one syntax form and never affect the Fixed-Point Exception Register. The andi. and andil. instructions copies the Summary Overflow (SO) bit from the Fixed-Point Exception Register into Condition Register Field 0 and sets one of the Less Than (LT), Greater Than (GT), or Equal To (EQ) bits of Condition Register Field 0.
Parameters

RA  Specifies target general-purpose register where result of operation is stored.
RS  Specifies source general-purpose register for operation.
UI  Specifies 16-bit unsigned integer for operation.

Examples

The following code logically ANDs the contents of GPR 4 with 0x0000 5730, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x7B41 92C0.
andi. 6,4,0x5730
# GPR 6 now contains 0x0000 1200.
# CRF 0 now contains 0x4.
```

Related Information

Fixed-Point Processor.
Fixed-Point Logical Instructions.

andis. or andiu. (AND Immediate Shifted) Instruction

Purpose

Logically ANDs the most significant 16 bits of the contents of a general-purpose register with a 16-bit unsigned integer and stores the result in a general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>29</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>UI</td>
</tr>
</tbody>
</table>

PowerPC

```
andis.     RA RS UI
```

POWER family

```
andiu.    RA RS UI
```

Description

The andis. and andiu. instructions logically AND the contents of general-purpose register (GPR) RS with the concatenation of a 16-bit unsigned integer, UI, and x’0000’ and then place the result into the target GPR RA.

The andis. and andiu. instructions have one syntax form and never affect the Fixed-Point Exception Register. The andis. and andiu. instructions set the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, or Summary Overflow (SO) bit in Condition Register Field 0.
Parameters

RA Specifies target general-purpose register where result of operation is stored.
RS Specifies source general-purpose register for operation.
UI Specifies 16-bit unsigned integer for operation.

Examples

The following code logically ANDs the contents of GPR 4 with 0x5730 0000, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

# Assume GPR 4 contains 0x7B41 92C0.
andis. 6,4,0x5730
# GPR 6 now contains 0x5300 0000.

Related Information

Fixed-Point Processor.
Fixed-Point Logical Instructions.

b (Branch) Instruction

Purpose

Branches to a specified target address.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>18</td>
</tr>
<tr>
<td>6-29</td>
<td>LL</td>
</tr>
<tr>
<td>30</td>
<td>AA</td>
</tr>
<tr>
<td>31</td>
<td>LK</td>
</tr>
</tbody>
</table>

\[ b \quad \text{target_address} \]
\[ ba \quad \text{target_address} \]
\[ bl \quad \text{target_address} \]
\[ bla \quad \text{target_address} \]

Description

The \( b \) instruction branches to an instruction specified by the branch target address. The branch target address is computed one of two ways.

Consider the following when using the \( b \) instruction:

- If the Absolute Address bit (AA) is 0, the branch target address is computed by concatenating the 24-bit \( LI \) field. This field is calculated by subtracting the address of the instruction from the target address and dividing the result by 4 and \( b'00' \). The result is then sign-extended to 32 bits and added to the address of this branch instruction.
- If the AA bit is 1, then the branch target address is the \( LI \) field concatenated with \( b'00' \) sign-extended to 32 bits. The \( LI \) field is the low-order 26 bits of the target address divided by four.
The *b* instruction has four syntax forms. Each syntax form has a different effect on the Link bit and Link Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Absolute Address Bit (AA)</th>
<th>Fixed-Point Exception Register</th>
<th>Link Bit (LK)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>0</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>ba</td>
<td>1</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>bl</td>
<td>0</td>
<td>None</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td>bla</td>
<td>1</td>
<td>None</td>
<td>1</td>
<td>None</td>
</tr>
</tbody>
</table>

The four syntax forms of the *b* instruction never affect the Fixed-Point Exception Register or Condition Register Field 0. The syntax forms set the AA bit and the Link bit (LK) and determine which method of calculating the branch target address is used. If the Link bit (LK) is set to 1, then the effective address of the instruction is placed in the Link Register.

**Parameters**

*target_address* Specifies the target address.

**Examples**

1. The following code transfers the execution of the program to *there*:

   ```
   here: b there
   cror 31,31,31
   # The execution of the program continues at there.
   there:
   ```

2. The following code transfers the execution of the program to *here* and sets the Link Register:

   ```
   bl here
   return: cror 31,31,31
   # The Link Register now contains the address of return.
   # The execution of the program continues at here.
   here:
   ```

**Related Information**

- [Branch Processor](#)
- [Branch Instructions](#)

**bc (Branch Conditional) Instruction**

**Purpose**

Conditionally branches to a specified target address.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>16</td>
</tr>
<tr>
<td>6-10</td>
<td>BO</td>
</tr>
<tr>
<td>11-15</td>
<td>BI</td>
</tr>
<tr>
<td>16-29</td>
<td>BD</td>
</tr>
</tbody>
</table>
### Description

The `bc` instruction branches to an instruction specified by the branch target address. The branch target address is computed one of two ways:

- If the Absolute Address bit (AA) is 0, then the branch target address is computed by concatenating the 14-bit Branch Displacement (BD) and b'00', sign-extending this to 32 bits, and adding the result to the address of this branch instruction.
- If the AA is 1, then the branch target address is BD concatenated with b'00' sign-extended to 32 bits.

The `bc` instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Absolute Address Bit (AA)</th>
<th>Fixed-Point Exception Register</th>
<th>Link Bit (LK)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bc</td>
<td>0</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>bca</td>
<td>1</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>bcl</td>
<td>0</td>
<td>None</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td>bcla</td>
<td>1</td>
<td>None</td>
<td>1</td>
<td>None</td>
</tr>
</tbody>
</table>

The four syntax forms of the `bc` instruction never affect the Fixed-Point Exception Register or Condition Register Field 0. The syntax forms set the AA bit and the Link bit (LK) and determine which method of calculating the branch target address is used. If the Link Bit (LK) is set to 1, then the effective address of the instruction is placed in the Link Register.

The Branch Option field (BO) is used to combine different types of branches into a single instruction. Extended mnemonics are provided to set the Branch Option field automatically.

The encoding for the BO field is defined in PowerPC architecture. The following list gives brief descriptions of the possible values for this field using pre-V2.00 encoding:

Table 31. BO Field Values Using pre-V2.00 Encoding

<table>
<thead>
<tr>
<th>BO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000y</td>
<td>Decrement the CTR; then branch if the decremented CTR is not 0 and the condition is False.</td>
</tr>
<tr>
<td>0001y</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0 and the condition is False.</td>
</tr>
<tr>
<td>001zy</td>
<td>Branch if the condition is False.</td>
</tr>
<tr>
<td>0100y</td>
<td>Decrement the CTR; then branch if bits the decremented CTR is not 0 and the condition is True.</td>
</tr>
<tr>
<td>0101y</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0 and the condition is True.</td>
</tr>
<tr>
<td>011zy</td>
<td>Branch if the condition is True.</td>
</tr>
<tr>
<td>1z00y</td>
<td>Decrement the CTR; then branch if the decremented CTR is not 0.</td>
</tr>
<tr>
<td>1z01y</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0.</td>
</tr>
<tr>
<td>1z1zz</td>
<td>Branch always.</td>
</tr>
</tbody>
</table>

See “Extended Mnemonics of Branch Instructions” on page 89 for more information.
In the PowerPC architecture, the bits are as follows:

- The z bit denotes a bit that must be 0. If the bit is not 0, the instruction form is invalid.
- The y bit provides a hint about whether a conditional branch is likely to be taken. The value of this bit can be either 0 or 1. The default value is 0.

In the POWER family architecture, the z and y bits can be either 0 or 1.

The encoding for the BO field using V2.00 encoding is briefly described below:

<table>
<thead>
<tr>
<th>BO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000z</td>
<td>Decrement the CTR; then branch if the decremented CTR is not 0 and the condition is False.</td>
</tr>
<tr>
<td>0001z</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0 and the condition is False.</td>
</tr>
<tr>
<td>001at</td>
<td>Branch if the condition is False.</td>
</tr>
<tr>
<td>0100z</td>
<td>Decrement the CTR; then branch if bits the decremented CTR is not 0 and the condition is True.</td>
</tr>
<tr>
<td>0101z</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0 and the condition is True.</td>
</tr>
<tr>
<td>011at</td>
<td>Branch if the condition is True.</td>
</tr>
<tr>
<td>1a00t</td>
<td>Decrement the CTR; then branch if the decremented CTR is not 0.</td>
</tr>
<tr>
<td>1a01t</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0.</td>
</tr>
<tr>
<td>1z1zz</td>
<td>Branch always.</td>
</tr>
</tbody>
</table>

The a and t bits of the BO field can be used by software to provide a hint about whether a branch is likely to be taken, as shown below:

<table>
<thead>
<tr>
<th>at</th>
<th>Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No hint is given.</td>
</tr>
<tr>
<td>01</td>
<td>Reserved</td>
</tr>
<tr>
<td>01</td>
<td>The branch is very likely not to be taken.</td>
</tr>
<tr>
<td>11</td>
<td>The branch is very likely to be taken.</td>
</tr>
</tbody>
</table>

### Parameters

- **target_address** Specifies the target address. For absolute branches such as bca and bcla, the target address can be immediate data containable in 16 bits.
- **BI** Specifies bit in Condition Register for condition comparison.
- **BO** Specifies Branch Option field used in instruction.

### Examples

The following code branches to a target address dependent on the value in the Count Register:

```
addi 8,0,3  # Loads GPR 8 with 0x3.
mtctr 8     # The Count Register (CTR) equals 0x3.
addic. 9,8,0x1 # Adds one to GPR 8 and places the result in GPR 9.
              # The Condition Register records a comparison against zero
              # with the result.
bc 0xC,0,there # Branch is taken if condition is true. 0 indicates that
              # the 0 bit in the Condition Register is checked to
              # determine if it is set (the LT bit is on). If it is set,
              # the branch is taken.
bcl 0x8,2,there
```
CTR is decremented by one, becoming 2.
The branch is taken if CTR is not equal to 0 and CTR bit 2
is set (the EQ bit is on).
The Link Register contains address of next instruction.

Related Information
Chapter 1, “Assembler Overview,” on page 1.
“Branch Processor” on page 19.
“Branch Instructions” on page 19.

bcctr or bcc (Branch Conditional to Count Register) Instruction

Purpose
Conditionally branches to the address contained within the Count Register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>19</td>
</tr>
<tr>
<td>6-10</td>
<td>BO</td>
</tr>
<tr>
<td>11-15</td>
<td>BI</td>
</tr>
<tr>
<td>16-18</td>
<td>///</td>
</tr>
<tr>
<td>19-20</td>
<td>BH</td>
</tr>
<tr>
<td>21-30</td>
<td>528</td>
</tr>
<tr>
<td>31</td>
<td>LK</td>
</tr>
</tbody>
</table>

PowerPC
bcctr
bcctrl

POWER family
bcc
bccl

See “Extended Mnemonics of Branch Instructions” on page 89 for more information.

Description
The bcctr and bcc instructions conditionally branch to an instruction specified by the branch target address contained within the Count Register. The branch target address is the concatenation of Count Register bits 0-29 and b'00'.

The bcctr and bcc instructions have two syntax forms. Each syntax form has a different effect on the Link bit and Link Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Absolute Address Bit (AA)</th>
<th>Fixed-Point Exception Register</th>
<th>Link Bit (LK)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcctr</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>


The two syntax forms of the `bcctr` and `bcc` instructions never affect the Fixed-Point Exception Register or Condition Register Field 0. If the Link bit is 1, then the effective address of the instruction following the branch instruction is placed into the Link Register.

The Branch Option field (BO) is used to combine different types of branches into a single instruction. Extended mnemonics are provided to set the Branch Option field automatically.

The encoding for the BO field is defined in PowerPC architecture. The following list gives brief descriptions of the possible values for this field using pre-V2.00 encoding:

<table>
<thead>
<tr>
<th>BO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000y</td>
<td>Decrement the CTR; then branch if the decremented CTR is not 0 and the condition is False.</td>
</tr>
<tr>
<td>0001y</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0 and the condition is False.</td>
</tr>
<tr>
<td>001zy</td>
<td>Branch if the condition is False.</td>
</tr>
<tr>
<td>0100y</td>
<td>Decrement the CTR; then branch if bits the decremented CTR is not 0 and the condition is True.</td>
</tr>
<tr>
<td>0101y</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0 and the condition is True.</td>
</tr>
<tr>
<td>011zy</td>
<td>Branch if the condition is True.</td>
</tr>
<tr>
<td>1000y</td>
<td>Decrement the CTR; then branch if the decremented CTR is not 0.</td>
</tr>
<tr>
<td>1010y</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0.</td>
</tr>
<tr>
<td>111zz</td>
<td>Branch always.</td>
</tr>
</tbody>
</table>

In the PowerPC architecture, the bits are as follows:
- The z bit denotes a bit that must be 0. If the bit is not 0, the instruction form is invalid.
- The y bit provides a hint about whether a conditional branch is likely to be taken. The value of this bit can be either 0 or 1. The default value is 0.

In the POWER family Architecture, the z and y bits can be either 0 or 1.

The encoding for the BO field using V2.00 encoding is briefly described below:

Table 33. BO Field Values Using V2.00 Encoding

<table>
<thead>
<tr>
<th>BO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000z</td>
<td>Decrement the CTR; then branch if the decremented CTR is not 0 and the condition is False.</td>
</tr>
<tr>
<td>0001z</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0 and the condition is False.</td>
</tr>
<tr>
<td>001at</td>
<td>Branch if the condition is False.</td>
</tr>
<tr>
<td>0100z</td>
<td>Decrement the CTR; then branch if bits the decremented CTR is not 0 and the condition is True.</td>
</tr>
<tr>
<td>0101z</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0 and the condition is True.</td>
</tr>
<tr>
<td>011at</td>
<td>Branch if the condition is True.</td>
</tr>
<tr>
<td>1000t</td>
<td>Decrement the CTR; then branch if the decremented CTR is not 0.</td>
</tr>
<tr>
<td>1010t</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0.</td>
</tr>
<tr>
<td>111zz</td>
<td>Branch always.</td>
</tr>
</tbody>
</table>

The a and t bits of the BO field can be used by software to provide a hint about whether a branch is likely to be taken, as shown below:

<table>
<thead>
<tr>
<th>at</th>
<th>Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No hint is given.</td>
</tr>
<tr>
<td>01</td>
<td>Reserved</td>
</tr>
<tr>
<td>01</td>
<td>The branch is very likely not to be taken.</td>
</tr>
</tbody>
</table>
The Branch Hint field (BH) is used to provide a hint about the use of the instruction, as shown below:

<table>
<thead>
<tr>
<th>BH</th>
<th>Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>The instruction is not a subroutine return; the target address is likely to be the same as the target address used the preceding time the branch was taken.</td>
</tr>
<tr>
<td>01</td>
<td>Reserved</td>
</tr>
<tr>
<td>10</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>The target address is not predictable.</td>
</tr>
</tbody>
</table>

**Parameters**

- **BO** Specifies Branch Option field.
- **BI** Specifies bit in Condition Register for condition comparison.
- **BIF** Specifies the Condition Register field that specifies the Condition Register bit (LT, GT, EQ, or SO) to be used for condition comparison.
- **BH** Provides a hint about the use of the instruction.

**Examples**

The following code branches from a specific address, dependent on a bit in the Condition Register, to the address contained in the Count Register, and no branch hints are given:

```assembly
bcctr 0x4,0,0
cror 31,31,31
# Branch occurs if LT bit in the Condition Register is 0.
# The branch will be to the address contained in
# the Count Register.
bcctr1 0xC,1,0
return: cror 31,31,31
# Branch occurs if GT bit in the Condition Register is 1.
# The branch will be to the address contained in
# the Count Register.
# The Link register now contains the address of return.
```

**Related Information**

- [Chapter 1, “Assembler Overview,” on page 1](#).
- “Branch Processor” on page 19.
- “Branch Instructions” on page 19.

**bclr or bcr (Branch Conditional Link Register) Instruction**

**Purpose**

Conditionally branches to an address contained in the Link Register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>19</td>
</tr>
<tr>
<td>6-10</td>
<td>BO</td>
</tr>
<tr>
<td>Bits</td>
<td>Value</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>11-15</td>
<td>BI</td>
</tr>
<tr>
<td>16-18</td>
<td>///</td>
</tr>
<tr>
<td>19-20</td>
<td>BH</td>
</tr>
<tr>
<td>21-30</td>
<td>16</td>
</tr>
<tr>
<td>31</td>
<td>LK</td>
</tr>
</tbody>
</table>

**PowerPC**

- `bclr` on page 151
- `bclrl` on page 151

**POWER family**

- `bcr` on page 151
- `bcrl` on page 151

See [“Extended Mnemonics of Branch Instructions” on page 89](#) for more information.

**Description**

The `bclr` and `bcr` instructions branch to an instruction specified by the branch target address. The branch target address is the concatenation of bits 0-29 of the Link Register and b'00'.

The `bclr` and `bcr` instructions have two syntax forms. Each syntax form has a different effect on the Link bit and Link Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Absolute Address Bit (AA)</th>
<th>Fixed-Point Exception Register</th>
<th>Link Bit (LK)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bclr</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>bclrl</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td>bcr</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>bcrl</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>None</td>
</tr>
</tbody>
</table>

The two syntax forms of the `bclr` and `bcr` instruction never affect the Fixed-Point Exception Register or Condition Register Field 0. If the Link bit (LK) is 1, then the effective address of the instruction that follows the branch instruction is placed into the Link Register.

The Branch Option field (BO) is used to combine different types of branches into a single instruction. Extended mnemonics are provided to set the Branch Option field automatically.

The encoding for the BO field is defined in PowerPC architecture. The following list gives brief descriptions of the possible values for this field:

<table>
<thead>
<tr>
<th>BO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000y</td>
<td>Decrement the CTR; then branch if the decremented CTR is not 0 and the condition is False.</td>
</tr>
<tr>
<td>0001y</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0 and the condition is False.</td>
</tr>
<tr>
<td>001zy</td>
<td>Branch if the condition is False.</td>
</tr>
<tr>
<td>0100y</td>
<td>Decrement the CTR; then branch if bits the decremented CTR is not 0 and the condition is True.</td>
</tr>
<tr>
<td>0101y</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0 and the condition is True.</td>
</tr>
<tr>
<td>011zy</td>
<td>Branch if the condition is True.</td>
</tr>
<tr>
<td>1200y</td>
<td>Decrement the CTR; then branch if the decremented CTR is not 0.</td>
</tr>
<tr>
<td>BO</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1z01y</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0.</td>
</tr>
<tr>
<td>1z1zz</td>
<td>Branch always.</td>
</tr>
</tbody>
</table>

In the PowerPC architecture, the bits are as follows:

- The z bit denotes a bit that must be 0. If the bit is not 0, the instruction form is invalid.
- The y bit provides a hint about whether a conditional branch is likely to be taken. The value of this bit can be either 0 or 1. The default value is 0.

In the POWER family Architecture, the z and y bits can be either 0 or 1.

The encoding for the BO field using V2.00 encoding is briefly described below:

Table 34. BO Field Values Using V2.00 Encoding

<table>
<thead>
<tr>
<th>BO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000z</td>
<td>Decrement the CTR; then branch if the decremented CTR is not 0 and the condition is False.</td>
</tr>
<tr>
<td>0001z</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0 and the condition is False.</td>
</tr>
<tr>
<td>001at</td>
<td>Branch if the condition is False.</td>
</tr>
<tr>
<td>0100z</td>
<td>Decrement the CTR; then branch if bits the decremented CTR is not 0 and the condition is True.</td>
</tr>
<tr>
<td>0101z</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0 and the condition is True.</td>
</tr>
<tr>
<td>011at</td>
<td>Branch if the condition is True.</td>
</tr>
<tr>
<td>1a00t</td>
<td>Decrement the CTR; then branch if the decremented CTR is not 0.</td>
</tr>
<tr>
<td>1a01t</td>
<td>Decrement the CTR; then branch if the decremented CTR is 0.</td>
</tr>
<tr>
<td>1z1zz</td>
<td>Branch always.</td>
</tr>
</tbody>
</table>

The a and t bits of the BO field can be used by software to provide a hint about whether a branch is likely to be taken, as shown below:

<table>
<thead>
<tr>
<th>at</th>
<th>Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No hint is given.</td>
</tr>
<tr>
<td>01</td>
<td>Reserved</td>
</tr>
<tr>
<td>01</td>
<td>The branch is very likely not to be taken.</td>
</tr>
<tr>
<td>11</td>
<td>The branch is very likely to be taken.</td>
</tr>
</tbody>
</table>

The Branch Hint field (BH) is used to provide a hint about the use of the instruction, as shown below:

<table>
<thead>
<tr>
<th>BH</th>
<th>Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>The instruction is not a subroutine return; the target address is likely to be the same as the target address used the preceding time the branch was taken.</td>
</tr>
<tr>
<td>01</td>
<td>Reserved</td>
</tr>
<tr>
<td>10</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>The target address is not predictable.</td>
</tr>
</tbody>
</table>

**Parameters**

- **BO** Specifies Branch Option field.
- **BI** Specifies bit in Condition Register for condition comparison.
- **BH** Provides a hint about the use of the instruction.
Examples
The following code branches to the calculated branch target address dependent on bit 0 of the Condition Register, and no branch hint is given:

    bc1r 0x0,0,0

# The Count Register is decremented.
# A branch occurs if the LT bit is set to zero in the
# Condition Register and if the Count Register
# does not equal zero.
# If the conditions are met, the instruction branches to
# the concatenation of bits 0-29 of the Link Register and b'00'.

Related Information
Chapter 1, “Assembler Overview,” on page 1.

“Branch Processor” on page 19.

“Branch Instructions” on page 19.

clc (Cache Line Compute Size) Instruction

Purpose
Places a specified cache line size in a general-purpose register.

Note: The clcs instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>531</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family
clcs [RT RA]

Description
The clcs instruction places the cache line size specified by RA into the target general-purpose register (GPR) RT. The value of RA determines the cache line size returned in GPR RT.

<table>
<thead>
<tr>
<th>Value of RA</th>
<th>Cache Line Size Returned in RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>00xxx</td>
<td>Undefined</td>
</tr>
<tr>
<td>010xx</td>
<td>Undefined</td>
</tr>
<tr>
<td>01100</td>
<td>Instruction Cache Line Size</td>
</tr>
<tr>
<td>01101</td>
<td>Data Cache Line Size</td>
</tr>
<tr>
<td>01110</td>
<td>Minimum Cache Line Size</td>
</tr>
<tr>
<td>01111</td>
<td>Maximum Cache Line Size</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>1xxxx</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

**Note:** The value in GPR $RT$ must lie between 64 and 4096, inclusive, or results will be undefined.

The `clcs` instruction has only one syntax form and does not affect the Fixed-Point Exception Register. If the Record (Rc) bit is set to 1, the Condition Register Field 0 is undefined.

**Parameters**

$RT$  Specifies target general-purpose register where result of operation is stored.

$RA$  Specifies cache line size requested.

**Examples**

The following code loads the maximum cache line size into GPR 4:

```plaintext
# Assume that 0xf is the cache line size requested.

clcs 4,0xf
```

# GPR 4 now contains the maximum Cache Line size.

**Related Information**

The `clf` (Cache Line Flush) instruction, `cli` (Cache Line Invalidate) instruction, `dcbst` (Data Cache Block Store) instruction, `dcbt` (Data Cache Block Touch) instruction, `dcbtst` (Data Cache Block Touch for Store) instruction, `dcbz` or `dclz` (Data Cache Block Set to Zero) instruction, `dcblst` (Data Cache Line Store) instruction, `icbi` (Instruction Cache Block Invalidate) instruction, `sync` (Synchronize) or `dcs` (Data Cache Synchronize) instruction.

**clf (Cache Line Flush) Instruction**

**Purpose**

Writes a line of modified data from the data cache to main memory, or invalidates cached instructions or unmodified data.

**Note:** The `clf` instruction is supported only in the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>118</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
**POWER family**

**clf**

### Description

The `clf` instruction calculates an effective address (EA) by adding the contents of general-purpose register (GPR) `RA` to the contents of GPR `RB`. If the `RA` field is 0, EA is the sum of the contents of `RB` and 0. If the `RA` field is not 0 and if the instruction does not cause a data storage interrupt, the result of the operation is placed back into GPR `RA`.

Consider the following when using the `clf` instruction:

- If the Data Relocate (DR) bit of the Machine State Register (MSR) is set to 0, the effective address is treated as a real address.
- If the MSR DR bit is set to 1, the effective address is treated as a virtual address. The MSR Instruction Relocate bit (IR) is ignored in this case.
- If a line containing the byte addressed by the EA is in the data cache and has been modified, writing the line to main memory is begun. If a line containing the byte addressed by EA is in one of the caches, the line is not valid.
- When MSR (DR) = 1, if the virtual address has no translation, a Data Storage interrupt occurs, setting the first bit of the Data Storage Interrupt Segment register to 1.
- A machine check interrupt occurs when the virtual address translates to an invalid real address and the line exists in the data cache.
- Address translation treats the instruction as a load to the byte addressed, ignoring protection and data locking. If this instruction causes a Translation Look-Aside buffer (TLB) miss, the reference bit is set.
- If the EA specifies an I/O address, the instruction is treated as a no-op, but the EA is placed in GPR `RA`.

The `clf` instruction has one syntax form and does not effect the Fixed-Point Exception register. If the Record (Rc) bit is set to 1, Condition Register Field 0 is undefined.

### Parameters

- **RA** Specifies the source general-purpose register for EA calculation and, if `RA` is not GPR 0, the target general-purpose register for operation.
- **RB** Specifies the source general-purpose register for EA calculation.

### Examples

The processor is not required to keep instruction storage consistent with data storage. The following code executes storage synchronization instructions prior to executing an modified instruction:

```assembly
# Assume that instruction A is assigned to storage location
# 0x0033 0020.

# Assume that the storage location to which A is assigned
# contains 0x0000 0000.

# Assume that GPR 3 contains 0x0000 0020.

# Assume that GPR 4 contains 0x0033 0020.

# Assume that GPR 5 contains 0x5000 0020.

st   R5, R4, R3     # Store branch instruction in memory
cif  R4, R3         # Flush A from cache to main memory
dcs                          # Ensure cif is complete
ics                          # Discard prefetched instructions
b 0x0033 0020         # Go execute the new instructions
```

After the store, but prior to the execution of the `clf`, `dcs`, and `ics` instructions, the copy of A in the cache contains the branch instruction. However, it is possible that the copy of A in main memory still contains 0.
The `clf` instruction copies the new instruction back to main memory and invalidates the cache line containing location A in both the instruction and data caches. The sequence of the `dcs` instruction followed by the `ics` instruction ensures that the new instruction is in main memory and that the copies of the location in the data and instruction caches are invalid before fetching the next instruction.

### Related Information

The `clcs` (Cache Line Compute Size) instruction, `cli` (Cache Line Invalidate) instruction, `dcbf` (Data Cache Block Flush) instruction, `dcbi` (Data Cache Block Invalidate) instruction, `dcbst` (Data Cache Block Store) instruction, `dcbt` (Data Cache Block Touch) instruction, `dcbtst` (Data Cache Block Touch for Store) instruction, `dcbs` or `dclz` (Data Cache Block Set to Zero) instruction, `dclst` (Data Cache Line Store) instruction, `icbi` (Instruction Cache Block Invalidate) instruction, `sync` (Synchronize) or `dcs` (Data Cache Synchronize) instruction.

### cli (Cache Line Invalidate) Instruction

#### Purpose

Invalidate a line containing the byte addressed in either the data or instruction cache, causing subsequent references to retrieve the line again from main memory.

**Note:** The `cli` instruction is supported only in the POWER family architecture.

#### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>502</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**POWER family**

`cli` `[RA] [RB]`

#### Description

The `cli` instruction invalidates a line containing the byte addressed in either the data or instruction cache. If `RA` is not 0, the `cli` instruction calculates an effective address (EA) by adding the contents of general-purpose register (GPR) `RA` to the contents of GPR `RB`. If `RA` is not GPR 0 or the instruction does not cause a Data Storage interrupt, the result of the calculation is placed back into GPR `RA`.

Consider the following when using the `cli` instruction:

- If the Data Relocate (DR) bit of the Machine State Register (MSR) is 0, the effective address is treated as a real address.
- If the MSR DR bit is 1, the effective address is treated as a virtual address. The MSR Relocate (IR) bit is ignored in this case.
- If a line containing the byte addressed by the EA is in the data or instruction cache, the line is made unusable so the next reference to the line is taken from main memory.
• When MSR (DR) = 1, if the virtual address has no translation, a Data Storage interrupt occurs, setting the first bit of the Data Storage Interrupt Segment Register to 1.
• Address translation treats the cli instruction as a store to the byte addressed, ignoring protection and data locking. If this instruction causes a Translation Look-Aside buffer (TLB) miss, the reference bit is set.
• If the EA specifies an I/O address, the instruction is treated as a no-op, but the EA is still placed in RA.

The cli instruction has only one syntax form and does not effect the Fixed-Point Exception Register. If the Record (Rc) bit is set to 1, the Condition Register Field 0 is undefined.

**Parameters**

RA  Specifies the source general-purpose register for EA calculation and possibly the target general-purpose register (when RA is not GPR 0) for operation.
RB  Specifies the source general-purpose register for EA calculation.

**Security**

The cli instruction is privileged.

**Related Information**

The clcs (Cache Line Compute Size) instruction, clf (Cache Line Flush) instruction, dcbf (Data Cache Block Flush) instruction, dcbi (Data Cache Block Invalidate) instruction, dcbst (Data Cache Block Store) instruction, dcbt (Data Cache Block Touch) instruction, dcbst (Data Cache Block Touch for Store) instruction, dcbz or dclz (Data Cache Block Set to Zero) instruction, dclst (Data Cache Line Store) instruction, icbi (Instruction Cache Block Invalidate) instruction, sync (Synchronize) or dcs (Data Cache Synchronize) instruction.

**Processing and Storage: Overview**

**cmp (Compare) Instruction**

**Purpose**

Compares the contents of two general-purpose registers algebraically.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-8</td>
<td>BF</td>
</tr>
<tr>
<td>9</td>
<td>/</td>
</tr>
<tr>
<td>10</td>
<td>L</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

cmp  BF  L  RA  RB
Description

The `cmp` instruction compares the contents of general-purpose register (GPR) RA with the contents of GPR RB as signed integers and sets one of the bits in Condition Register Field BF.

BF can be Condition Register Field 0-7; programmers can specify which Condition Register Field will indicate the result of the operation.

The bits of Condition Register Field BF are interpreted as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LT</td>
<td>(RA) &lt; SI</td>
</tr>
<tr>
<td>1</td>
<td>GT</td>
<td>(RA) &gt; SI</td>
</tr>
<tr>
<td>2</td>
<td>EQ</td>
<td>(RA) = SI</td>
</tr>
<tr>
<td>3</td>
<td>SO</td>
<td>SO,OV</td>
</tr>
</tbody>
</table>

The `cmp` instruction has one syntax form and does not affect the Fixed-Point Exception Register. Condition Register Field 0 is unaffected unless it is specified as BF by the programmer.

Parameters

BF    Specifies Condition Register Field 0-7 which indicates result of compare.
L     Must be set to 0 for the 32-bit subset architecture.
RA    Specifies source general-purpose register for operation.
RB    Specifies source general-purpose register for operation.

Examples

The following code compares the contents of GPR 4 and GPR 6 as signed integers and sets Condition Register Field 0 to reflect the result of the operation:

```c

# Assume GPR 4 contains 0xFFFF FFE7.
# Assume GPR 5 contains 0x0000 0011.
# Assume 0 is Condition Register Field 0.
cmp 0,4,6
# The LT bit of Condition Register Field 0 is set.
```

Related Information

The `cmp` (Compare Immediate) instruction, `cmpl` (Compare Logical) instruction, `cmpli` (Compare Logical Immediate) instruction.

Fixed-Point Processor.

**cmpi (Compare Immediate) Instruction**

**Purpose**

Compares the contents of a general-purpose register and a given value algebraically.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>11</td>
</tr>
<tr>
<td>6-8</td>
<td>BF</td>
</tr>
<tr>
<td>9</td>
<td>/</td>
</tr>
<tr>
<td>10</td>
<td>L</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>SI</td>
</tr>
</tbody>
</table>

`cmpi`  

See [Extended Mnemonics of Fixed-Point Compare Instructions](#) for more information.

**Description**

The `cmpi` instruction compares the contents of general-purpose register (GPR) `RA` and a 16-bit signed integer, `SI`, as signed integers and sets one of the bits in Condition Register Field `BF`.

`BF` can be Condition Register Field 0-7; programmers can specify which Condition Register Field will indicate the result of the operation.

The bits of Condition Register Field `BF` are interpreted as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LT</td>
<td>(RA) &lt; SI</td>
</tr>
<tr>
<td>1</td>
<td>GT</td>
<td>(RA) &gt; SI</td>
</tr>
<tr>
<td>2</td>
<td>EQ</td>
<td>(RA) = SI</td>
</tr>
<tr>
<td>3</td>
<td>SO</td>
<td>SO,OV</td>
</tr>
</tbody>
</table>

The `cmpi` instruction has one syntax form and does not affect the Fixed-Point Exception Register. Condition Register Field 0 is unaffected unless it is specified as `BF` by the programmer.

**Parameters**

- `BF` Specifies Condition Register Field 0-7 which indicates result of compare.
- `L` Must be set to 0 for the 32-bit subset architecture.
- `RA` Specifies first source general-purpose register for operation.
- `SI` Specifies 16-bit signed integer for operation.

**Examples**

The following code compares the contents of GPR 4 and the signed integer 0x11 and sets Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xFFFF FFE7.
cmpi 0,4,0x11
# The LT bit of Condition Register Field 0 is set.
```
Related Information
The \texttt{cmp} (Compare) instruction, \texttt{cmpl} (Compare Logical) instruction, \texttt{cmpli} (Compare Logical Immediate) instruction.

Fixed-Point Processor.

\textbf{cmpl (Compare Logical) Instruction}

\textbf{Purpose}
Compares the contents of two general-purpose registers logically.

\textbf{Syntax}

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-8</td>
<td>BF</td>
</tr>
<tr>
<td>9</td>
<td>/</td>
</tr>
<tr>
<td>10</td>
<td>L</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>32</td>
</tr>
</tbody>
</table>

\texttt{cmpl BF, L, RA, RB}

See \textit{Extended Mnemonics of Fixed-Point Compare Instructions} for more information.

\textbf{Description}
The \texttt{cmpl} instruction compares the contents of general-purpose register (GPR) \texttt{RA} with the contents of GPR \texttt{RB} as unsigned integers and sets one of the bits in Condition Register Field \texttt{BF}.

\texttt{BF} can be Condition Register Field 0-7; programmers can specify which Condition Register Field will indicate the result of the operation.

The bits of Condition Register Field \texttt{BF} are interpreted as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LT</td>
<td>(RA) &lt; SI</td>
</tr>
<tr>
<td>1</td>
<td>GT</td>
<td>(RA) &gt; SI</td>
</tr>
<tr>
<td>2</td>
<td>EQ</td>
<td>(RA) = SI</td>
</tr>
<tr>
<td>3</td>
<td>SO</td>
<td>SO,OV</td>
</tr>
</tbody>
</table>

The \texttt{cmpl} instruction has one syntax form and does not affect the Fixed-Point Exception Register. Condition Register Field 0 is unaffected unless it is specified as \texttt{BF} by the programmer.
Parameters

**BF**  Specifies Condition Register Field 0-7 which indicates result of compare.
**L**   Must be set to 0 for the 32-bit subset architecture.
**RA**  Specifies source general-purpose register for operation.
**RB**  Specifies source general-purpose register for operation.

Examples
The following code compares the contents of GPR 4 and GPR 5 as unsigned integers and sets Condition Register Field 0 to reflect the result of the operation:

```plaintext
# Assume GPR 4 contains 0xFFFF 0000.
# Assume GPR 5 contains 0x7FFF 0000.
# Assume 0 is Condition Register Field 0.
cmpl 0,4,5
# The GT bit of Condition Register Field 0 is set.
```

Related Information
The **cmp** (Compare) instruction, **cmpi** (Compare Immediate) instruction, **cmpli** (Compare Logical Immediate) instruction.

---

**cmpli** (Compare Logical Immediate) Instruction

Purpose
Compares the contents of a general-purpose register and a given value logically.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>10</td>
</tr>
<tr>
<td>6-8</td>
<td>BF</td>
</tr>
<tr>
<td>9</td>
<td>/</td>
</tr>
<tr>
<td>10</td>
<td>L</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>UI</td>
</tr>
</tbody>
</table>

**cmpli**

See [Extended Mnemonics of Fixed-Point Compare Instructions](#) for more information.

Description
The **cmpli** instruction compares the contents of general-purpose register (GPR) **RA** with the concatenation of `x'0000'` and a 16-bit unsigned integer, **UI**, as unsigned integers and sets one of the bits in the Condition Register Field **BF**.

**BF** can be Condition Register Field 0-7; programmers can specify which Condition Register Field will indicate the result of the operation.
The bits of Condition Register Field \( BF \) are interpreted as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LT</td>
<td>(RA) &lt; SI</td>
</tr>
<tr>
<td>1</td>
<td>GT</td>
<td>(RA) &gt; SI</td>
</tr>
<tr>
<td>2</td>
<td>EQ</td>
<td>(RA) = SI</td>
</tr>
<tr>
<td>3</td>
<td>SO</td>
<td>SO,OV</td>
</tr>
</tbody>
</table>

The `cmpli` instruction has one syntax form and does not affect the Fixed-Point Exception Register. Condition Register Field 0 is unaffected unless it is specified as \( BF \) by the programmer.

**Parameters**

- \( BF \) Specifies Condition Register Field 0-7 that indicates result of compare.
- \( L \) Must be set to 0 for the 32-bit subset architecture.
- \( RA \) Specifies source general-purpose register for operation.
- \( UI \) Specifies 16-bit unsigned integer for operation.

**Examples**

The following code compares the contents of GPR 4 and the unsigned integer 0xff and sets Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x0000 00ff.
cmpli 0,4,0xff
# The EQ bit of Condition Register Field 0 is set.
```

**Related Information**

The `cmp` (Compare) instruction, `cmpi` (Compare Immediate) instruction, `cmpl` (Compare Logical) instruction.

[Fixed-Point Processor](#)

**cntlz (Count Leading Zeros Double Word) Instruction**

**Purpose**

Count the number of consecutive zero bits in the contents of a general purpose register, beginning with the high-order bit.

This instruction should only be used on 64-bit PowerPC processors running a 64-bit application.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>00000</td>
</tr>
<tr>
<td>21-30</td>
<td>58</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
**Description**

A count of the number of consecutive zero bits, starting at bit 0 (the high-order bit) of register GPR RS is placed into GPR RA. This number ranges from 0 to 64, inclusive.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

Condition Register (CR0 field):

Affected: LT, GT, EQ, SO (if Rc = 1)

Note: If Rc = 1, then LT is cleared in the CR0 field.

**Parameters**

RA  Specifies the target general purpose register for the results of the instruction.

RS  Specifies the source general purpose register containing the double-word to examine.

**Implementation**

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

---

### cntlzw or cntlz (Count Leading Zeros Word) Instruction

**Purpose**

Placed the number of leading zeros from a source general-purpose register in a general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>26</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC

cntlzw  

cntlzw.
POWER family

cntlz  RA RS
cntlz.  RA RS

Description

The **cntlzw** and **cntlz** instructions count the number (between 0 and 32 inclusive) of consecutive zero bits starting at bit 0 of general-purpose register (GPR) RS and store the result in the target GPR RA.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cntlzw</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>cntlzw.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>cntlz</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>cntlz.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the **cntlzw** instruction and the two syntax forms of the **cntlz** instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

* RA  Specifies target general-purpose register where result of operation is stored.
* RS  Specifies source general-purpose register for operation.

Examples

The following code counts the number of leading zeros in the value contained in GPR 3 and places the result back in GPR 3:

```plaintext
# Assume GPR 3 contains 0x0061 9920.
cntlzw 3,3
# GPR 3 now holds 0x0000 0009.
```

Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Logical Instructions](#)

**crand (Condition Register AND) Instruction**

Purpose

Places the result of ANDing two Condition Register bits in a Condition Register bit.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>19</td>
</tr>
<tr>
<td>6-10</td>
<td>BT</td>
</tr>
<tr>
<td>11-15</td>
<td>BA</td>
</tr>
</tbody>
</table>
The `crand` instruction logically ANDs the Condition Register bit specified by `BA` and the Condition Register bit specified by `BB` and places the result in the target Condition Register bit specified by `BT`.

The `crand` instruction has one syntax form and does not affect the Fixed-Point Exception Register.

### Parameters

- **`BT`**: Specifies target Condition Register bit where result of operation is stored.
- **`BA`**: Specifies source Condition Register bit for operation.
- **`BB`**: Specifies source Condition Register bit for operation.

### Examples

The following code logically ANDs Condition Register bits 0 and 5 and stores the result in Condition Register bit 31:

```assembly
# Assume Condition Register bit 0 is 1.
# Assume Condition Register bit 5 is 0.
crand 31,0,5
# Condition Register bit 31 is now 0.
```

### Related Information

- [Branch Processor](#).
- [Condition Register Instructions](#).

---

### crandc (Condition Register AND with Complement) Instruction

#### Purpose

Places the result of ANDing one Condition Register bit and the complement of a Condition Register bit in a Condition Register bit.

#### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>19</td>
</tr>
<tr>
<td>6-10</td>
<td>BT</td>
</tr>
<tr>
<td>11-15</td>
<td>BA</td>
</tr>
<tr>
<td>16-20</td>
<td>BB</td>
</tr>
<tr>
<td>21-30</td>
<td>129</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>
The `crandc` instruction logically ANDs the Condition Register bit specified in `BA` and the complement of the Condition Register bit specified by `BB` and places the result in the target Condition Register bit specified by `BT`.

The `crandc` instruction has one syntax form and does not affect the Fixed-Point Exception Register.

**Parameters**

- **BT**: Specifies target Condition Register bit where result of operation is stored.
- **BA**: Specifies source Condition Register bit for operation.
- **BB**: Specifies source Condition Register bit for operation.

**Examples**

The following code logically ANDs Condition Register bit 0 and the complement of Condition Register bit 5 and puts the result in bit 31:

```plaintext
# Assume Condition Register bit 0 is 1.
# Assume Condition Register bit 5 is 0.
crandc 31,0,5
```

# Condition Register bit 31 is now 1.

**Related Information**

- Branch Processor
- Condition Register Instructions

---

### creqv (Condition Register Equivalent) Instruction

**Purpose**

Places the complemented result of XORing two Condition Register bits in a Condition Register bit.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>19</td>
</tr>
<tr>
<td>6-10</td>
<td>BT</td>
</tr>
<tr>
<td>11-15</td>
<td>BA</td>
</tr>
<tr>
<td>16-20</td>
<td>BB</td>
</tr>
<tr>
<td>21-30</td>
<td>289</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

`creqv`  

See Extended Mnemonics of Condition Register Logical Instructions for more information.
Description

The `creqv` instruction logically XORs the Condition Register bit specified in \( BA \) and the Condition Register bit specified by \( BB \) and places the complemented result in the target Condition Register bit specified by \( BT \).

The `creqv` instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters

- \( BT \) Specifies target Condition Register bit where result of operation is stored.
- \( BA \) Specifies source Condition Register bit for operation.
- \( BB \) Specifies source Condition Register bit for operation.

Examples

The following code places the complemented result of XORing Condition Register bits 8 and 4 into Condition Register bit 4:

```
# Assume Condition Register bit 8 is 1.
# Assume Condition Register bit 4 is 0.
creqv 4,8,4
# Condition Register bit 4 is now 0.
```

Related Information

- Branch Processor
- Condition Register Instructions

---

**crnand (Condition Register NAND) Instruction**

**Purpose**

Places the complemented result of ANDing two Condition Register bits in a Condition Register bit.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>19</td>
</tr>
<tr>
<td>6-10</td>
<td>( BT )</td>
</tr>
<tr>
<td>11-15</td>
<td>( BA )</td>
</tr>
<tr>
<td>16-20</td>
<td>( BB )</td>
</tr>
<tr>
<td>21-30</td>
<td>225</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

```
crnand \( BT \ BA \ BB \)
```

**Description**

The `crnand` instruction logically ANDs the Condition Register bit specified by \( BA \) and the Condition Register bit specified by \( BB \) and places the complemented result in the target Condition Register bit specified by \( BT \).

The `crnand` instruction has one syntax form and does not affect the Fixed-Point Exception Register.
Parameters

- **BT**: Specifies target Condition Register bit where result of operation is stored.
- **BA**: Specifies source Condition Register bit for operation.
- **BB**: Specifies source Condition Register bit for operation.

Examples

The following code logically ANDs Condition Register bits 8 and 4 and places the complemented result into Condition Register bit 4:

```plaintext
# Assume Condition Register bit 8 is 1.
# Assume Condition Register bit 4 is 0.
crnand 4,8,4
# Condition Register bit 4 is now 1.
```

Related Information

- [Branch Processor](#)
- [Condition Register Instructions](#)

**crnor** (Condition Register NOR) Instruction

**Purpose**

Places the complemented result of ORing two Condition Register bits in a Condition Register bit.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>19</td>
</tr>
<tr>
<td>6-10</td>
<td>BT</td>
</tr>
<tr>
<td>11-15</td>
<td>BA</td>
</tr>
<tr>
<td>16-20</td>
<td>BB</td>
</tr>
<tr>
<td>21-30</td>
<td>33</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

```
crnor BT BA BB
```

See [Extended Mnemonics of Condition Register Logical Instructions](#) for more information.

**Description**

The **crnor** instruction logically ORs the Condition Register bit specified in **BA** and the Condition Register bit specified by **BB** and places the complemented result in the target Condition Register bit specified by **BT**.

The **crnor** instruction has one syntax form and does not affect the Fixed Point Exception Register.

**Parameters**

- **BT**: Specifies target Condition Register bit where result of operation is stored.
- **BA**: Specifies source Condition Register bit for operation.
**Examples**
The following code logically ORs Condition Register bits 8 and 4 and stores the complemented result into Condition Register bit 4:

```
# Assume Condition Register bit 8 is 1.
# Assume Condition Register bit 4 is 0.
cror 4,8,4
# Condition Register bit 4 is now 0.
```

**Related Information**
- [Branch Processor](#)
- [Condition Register Instructions](#)

---

**cror (Condition Register OR) Instruction**

**Purpose**
Places the result of ORing two Condition Register bits in a Condition Register bit.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>19</td>
</tr>
<tr>
<td>6-10</td>
<td>BT</td>
</tr>
<tr>
<td>11-15</td>
<td>BA</td>
</tr>
<tr>
<td>16-20</td>
<td>BB</td>
</tr>
<tr>
<td>21-30</td>
<td>449</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

```
cror [BT][BA][BB]
```

See [Extended Mnemonics of Condition Register Logical Instructions](#) for more information.

**Description**
The **cror** instruction logically ORs the Condition Register bit specified by **BA** and the Condition Register bit specified by **BB** and places the result in the target Condition Register bit specified by **BT**.

The **cror** instruction has one syntax form and does not affect the Fixed-Point Exception Register.

**Parameters**

- **BT**  Specified target Condition Register bit where result of operation is stored.
- **BA**  Specified source Condition Register bit for operation.
- **BB**  Specified source Condition Register bit for operation.

**Examples**
The following code places the result of ORing Condition Register bits 8 and 4 into Condition Register bit 4:
# Assume Condition Register bit 8 is 1.
# Assume Condition Register bit 4 is 0.
crorc 4,8,4
# Condition Register bit 4 is now 1.

**Related Information**
- [Branch Processor](#)
- [Condition Register Instructions](#)

## crorc (Condition Register OR with Complement) Instruction

### Purpose
Places the result of ORing a Condition Register bit and the complement of a Condition Register bit in a Condition Register bit.

### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>19</td>
</tr>
<tr>
<td>6-10</td>
<td>BT</td>
</tr>
<tr>
<td>11-15</td>
<td>BA</td>
</tr>
<tr>
<td>16-20</td>
<td>BB</td>
</tr>
<tr>
<td>21-30</td>
<td>417</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

crorc <BT BA BB>

### Description
The **crorc** instruction logically ORs the Condition Register bit specified by *BA* and the complement of the Condition Register bit specified by *BB* and places the result in the target Condition Register bit specified by *BT*.

The **crorc** instruction has one syntax form and does not affect the Fixed-Point Exception Register.

### Parameters

- **BT** Specifies target Condition Register bit where result of operation is stored.
- **BA** Specifies source Condition Register bit for operation.
- **BB** Specifies source Condition Register bit for operation.

### Examples
The following code places the result of ORing Condition Register bit 8 and the complement of Condition Register bit 4 into Condition Register bit 4:

```plaintext
# Assume Condition Register bit 8 is 1.
# Assume Condition Register bit 4 is 0.
crorc 4,8,4
# Condition Register bit 4 is now 1.
```
crxor (Condition Register XOR) Instruction

Purpose
Places the result of XORing two Condition Register bits in a Condition Register bit.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>19</td>
</tr>
<tr>
<td>6-10</td>
<td>BT</td>
</tr>
<tr>
<td>11-15</td>
<td>BA</td>
</tr>
<tr>
<td>16-20</td>
<td>BB</td>
</tr>
<tr>
<td>21-30</td>
<td>193</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

\texttt{crxor \: BT\: BA\: BB}

See [Extended Mnemonics of Condition Register Logical Instructions](#) for more information.

Description
The \texttt{crxor} instruction logically XORs the Condition Register bit specified by \texttt{BA} and the Condition Register bit specified by \texttt{BB} and places the result in the target Condition Register bit specified by \texttt{BT}.

The \texttt{crxor} instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters

- \texttt{BT} Specifies target Condition Register bit where result of operation is stored.
- \texttt{BA} Specifies source Condition Register bit for operation.
- \texttt{BB} Specifies source Condition Register bit for operation.

Examples
The following code places the result of XORing Condition Register bits 8 and 4 into Condition Register bit 4:

\begin{verbatim}
# Assume Condition Register bit 8 is 1.
# Assume Condition Register bit 4 is 1.
cr xor 4,8,4
# Condition Register bit 4 is now 0.
\end{verbatim}
**dcbf (Data Cache Block Flush) Instruction**

**Purpose**
Copies modified cache blocks to main storage and invalidates the copy in the data cache.

*Note:* The *dcbf* instruction is supported only in the PowerPC architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>86</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

*PowerPC dcbf RA RB*

**Description**
The *dcbf* instruction calculates an effective address (EA) by adding the contents of general-purpose register (GPR) *RA* to the contents of GPR *RB*. If the *RA* field is 0, EA is the sum of the contents of *RB* and 0. If the cache block containing the target storage locations is in the data cache, it is copied back to main storage, provided it is different than the main storage copy.

Consider the following when using the *dcbf* instruction:
- If a block containing the byte addressed by the EA is in the data cache and has been modified, the block is copied to main memory. If a block containing the byte addressed by EA is in one of the caches, the block is made not valid.
- If the EA specifies a direct store segment address, the instruction is treated as a no-op.

The *dcbf* instruction has one syntax form and does not effect the Fixed-Point Exception Register.

**Parameters**
- *RA* Specifies the source general-purpose register for operation.
- *RB* Specifies the source general-purpose register for operation.

**Examples**
The software manages the coherency of storage shared by the processor and another system component, such as an I/O device that does not participate in the storage coherency protocol. The following code flushes the shared storage from the data cache prior to allowing another system component access to the storage:

```
# Assume that the variable A is assigned to storage location
# 0x0000 4540.
# Assume that the storage location to which A is assigned
# contains 0.
# Assume that GPR 3 contains 0x0000 0040.
```
# Assume that GPR 4 contains 0x0000 4500.
# Assume that GPR 5 contains -1.

```
st R5,R4,R3          # Store 0xFFFF FFFF to A
dcbf R4,R3           # Flush A from cache to main memory
sync                  # Ensure dcbf is complete. Start I/O operation
```

After the store, but prior to the execution of the `dcbf` and `sync` instructions, the copy of A in the cache contains a -1. However, it is possible that the copy of A in main memory still contains 0. After the `sync` instruction completes, the location to which A is assigned in main memory contains -1 and the processor data cache no longer contains a copy of location A.

## Related Information

The `clcs` (Cache Line Compute Size) instruction, `clf` (Cache Line Flush) instruction, `cli` (Cache Line Invalidate) instruction, `dcbi` (Data Cache Block Invalidate) instruction, `dcbst` (Data Cache Block Store) instruction, `dcbft` (Data Cache Block Touch) instruction, `dcbst` (Data Cache Block Touch for Store) instruction, `dcby` or `dcz` (Data Cache Block Set to Zero) instruction, `dcist` (Data Cache Line Store) instruction, `incbi` (Instruction Cache Block Invalidate) instruction, `sync` (Synchronize) or `dcs` (Data Cache Synchronize) instruction.

### dcbi (Data Cache Block Invalidate) Instruction

#### Purpose

Invalidates a block containing the byte addressed in the data cache, causing subsequent references to retrieve the block again from main memory.

**Note:** The dcbi instruction is supported only in the PowerPC architecture.

#### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>//</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>470</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

**PowerPC**

```
dcbi  RA RB
```

#### Description

If the contents of general-purpose register (GPR) RA is not 0, the dcbi instruction computes an effective address (EA) by adding the contents of GPR RA to the contents of GPR RB. Otherwise, the EA is the content of GPR RB.

If the cache block containing the addressed byte is in the data cache, the block is made invalid. Subsequent references to a byte in the block cause a reference to main memory.

The dcbi instruction is treated as a store to the addressed cache block with respect to protection.
The dcbi instruction has only one syntax form and does not effect the Fixed-Point Exception register.

**Parameters**

RA  Specifies the source general-purpose register for EA computation.
RB  Specifies the source general-purpose register for EA computation.

**Security**

The dcbi instruction is privileged.

**Related Information**

The clcs (Cache Line Compute Size) instruction, clf (Cache Line Flush) instruction, cli (Cache Line Invalidate) instruction, dcbf (Data Cache Block Flush) instruction, dcbst (Data Cache Block Store) instruction, dcbt (Data Cache Block Touch) instruction, dcbzt (Data Cache Block Touch for Store) instruction, dcbz (or dclz (Data Cache Block Set to Zero) instruction, dclst (Data Cache Line Store) instruction, icbi (Instruction Cache Block Invalidate) instruction, sync (Synchronize) or dcs (Data Cache Synchronize) instruction.

**Processing and Storage**

**dcbst (Data Cache Block Store) Instruction**

**Purpose**

Allows a program to copy the contents of a modified block to main memory.

Note: The dcbst instruction is supported only in the PowerPC architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>54</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

PowerPC
dcbst  RA  RB

**Description**

The dcbst instruction causes any modified copy of the block to be copied to main memory. If RA is not 0, the dcbst instruction computes an effective address (EA) by adding the contents of general-purpose register (GPR) RA to the contents of GPR RB. Otherwise, the EA is the contents of RB. If the cache block containing the addressed byte is in the data cache and is modified, the block is copied to main memory.
The `dcbst` instruction may be used to ensure that the copy of a location in main memory contains the most recent updates. This may be important when sharing memory with an I/O device that does not participate in the coherence protocol. In addition, the `dcbst` instruction can ensure that updates are immediately copied to a graphics frame buffer.

Treat the `dcbst` instruction as a load from the addressed byte with respect to address translation and protection.

The `dcbst` instruction has one syntax form and does not effect the Fixed-Point Exception register.

**Parameters**

- **RA** Specifies the source general-purpose register for EA computation.
- **RB** Specifies the source general-purpose register for EA computation.

**Examples**

1. The following code shares memory with an I/O device that does not participate in the coherence protocol:

   ```
   # Assume that location A is memory that is shared with the I/O device.
   # Assume that GPR 2 contains a control value indicating that and I/O operation should start.
   # Assume that GPR 3 contains the new value to be placed in location A.
   # Assume that GPR 4 contains the address of location A.
   # Assume that GPR 5 contains the address of a control register in the I/O device.
   st 3,0,4  # Update location A.
   dcbst 0,4  # Copy new content of location A and other bytes in cache block to main memory.
   sync     # Ensure the dcbst instruction has completed.
   st 2,0,5  # Signal I/O device that location A has been update.
   ```

2. The following code copies to a graphics frame buffer, ensuring that new values are displayed without delay:

   ```
   # Assume that target memory is a graphics frame buffer.
   # Assume that GPR 2, 3, and 4 contain new values to be displayed.
   # Assume that GPR 5 contains the address minus 4 of where the first value is to be stored.
   # Assume that the 3 target locations are known to be in a single cache block.
   addi 6,5,4  # Compute address of first memory location.
   stwu 2,4(5) # Store value and update address ptr.
   stwu 3,4(5) # Store value and update address ptr.
   stwu 4,4(5) # Store value and update address ptr.
   dcbst 0,6   # Copy new content of cache block to frame buffer. New values are displayed.
   ```

**Related Information**

The `clcs` (Cache Line Compute Size) instruction, `clf` (Cache Line Flush) instruction, `cli` (Cache Line Invalidate) instruction, `dcbf` (Data Cache Block Flush) instruction, `dcbi` (Data Cache Block Invalidate) instruction, `dcbt` (Data Cache Block Touch) instruction, `dcbtst` (Data Cache Block Touch for Store) instruction, `dcbz` or `dclz` (Data Cache Block Set to Zero) instruction, `dclst` (Data Cache Line Store) instruction, `icbi` (Instruction Cache Block Invalidate) instruction, `sync` (Synchronize) or `dcs` (Data Cache Synchronize) instruction.
dcbt (Data Cache Block Touch) Instruction

Purpose
Allows a program to request a cache block fetch before it is actually needed by the program.

Note: The dcbt instruction is supported only in the POWER5 architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7-10</td>
<td>TH</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>278</td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

Description
The dcbt instruction may improve performance by anticipating a load from the addressed byte. The block containing the byte addressed by the effective address (EA) is fetched into the data cache before the block is needed by the program. The program can later perform loads from the block and may not experience the added delay caused by fetching the block into the cache. Executing the dcbt instruction does not invoke the system error handler.

If general-purpose register (GPR) RA is not 0, the effective address (EA) is the sum of the content of GPR RA and the content of GPR RB. Otherwise, the EA is the content of GPR RB.

Consider the following when using the dcbt instruction:
• If the EA specifies a direct store segment address, the instruction is treated as a no-op.
• The access is treated as a load from the addressed cache block with respect to protection. If protection does not permit access to the addressed byte, the dcbt instruction performs no operations.

Note: If a program needs to store to the data cache block, use the dcbtst (Data Cache Block Touch for Store) instruction.

The Touch Hint field (TH) is used to provide a hint that the program will probably load soon from the storage locations specified by the EA and the TH field. The hint is ignored for locations that are caching-inhibited or guarded. The encodings of the TH field are as follows:

<table>
<thead>
<tr>
<th>TH</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>The program will probably soon load from the byte addressed by EA.</td>
</tr>
<tr>
<td>0001</td>
<td>The program will probably soon load from the data stream consisting of the block containing the byte addressed by EA and an unlimited number of sequentially following blocks (that is, the blocks containing the bytes addressed by EA + n * block_size, where n = 0, 1, 2,...).</td>
</tr>
</tbody>
</table>
The program will probably soon load from the data stream consisting of the block containing the byte addressed by EA and an unlimited number of sequentially preceding blocks (that is, the blocks containing the bytes addressed by EA - n * block_size, where n = 0, 1, 2...).

The `dcbt` instruction provides a hint that describes certain attributes of a data stream, and optionally indicates that the program will probably soon load from the stream. The EA is interpreted as described in Table 35.

The `dcbt` instruction provides a hint that describes certain attributes of a data stream, or indicates that the program will probably soon load from data streams that have been described using `dcbt` instructions in which TH[0] = 1 or probably no longer load from such data streams. The EA is interpreted as described in Table 36.

The `dcbt` instruction serves as both a basic and extended mnemonic. The `dcbt` mnemonic with three operands is the basic form, and the `dcbt` with two operands is the extended form. In the extended form, the TH field is omitted and assumed to be 0b0000.

### Table 35. EA Encoding when TH=0b1000

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-56</td>
<td>EA_TRUNC</td>
<td>High-order 57 bits of the effective address of the first unit of the data stream.</td>
</tr>
<tr>
<td>57</td>
<td>D</td>
<td>Direction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: Subsequent units are the sequentially following units.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Subsequent units are the sequentially preceding units.</td>
</tr>
<tr>
<td>58</td>
<td>UG</td>
<td>No information is provided by the UG field.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: The number of units in the data stream is unlimited, the program's need for each block of the stream is not likely to be transient, and the program will probably soon load from the stream.</td>
</tr>
<tr>
<td>59</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>60–63</td>
<td>ID</td>
<td>Stream ID to use for this stream.</td>
</tr>
</tbody>
</table>

### Table 36. EA Encoding when TH=0b1010

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-31</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>32</td>
<td>GO</td>
<td>No information is provided by the GO field</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: The program will probably soon load from all nascent data streams that have been completely described, and will probably no longer load from all other data streams.</td>
</tr>
<tr>
<td>33-34</td>
<td>S</td>
<td>Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00: No information is provided by the S field.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01: Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10: The program will probably no longer load from the stream associated with the Stream ID (all other fields of the EA are ignored except for the ID field).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11: The program will probably no longer load from the data streams associated with all stream IDs (all other fields of the EA are ignored).</td>
</tr>
<tr>
<td>35-46</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Table 36. EA Encoding when TH=0b1010  (continued)

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>47-56</td>
<td>UNIT_CNT</td>
<td>Number of units in the data stream.</td>
</tr>
<tr>
<td>57</td>
<td>T</td>
<td>0  No information is provided by the T field.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1  The program’s need for each block of the data stream is likely to be transient (that is, the time interval during which the program accesses the block is likely to be short).</td>
</tr>
<tr>
<td>58</td>
<td>U</td>
<td>0  No information is provided by the U field.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1  The number of units in the data stream is unlimited (and the UNIT_CNT field is ignored).</td>
</tr>
<tr>
<td>59</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>60-63</td>
<td>ID</td>
<td>Stream ID to use for this stream.</td>
</tr>
</tbody>
</table>

The `dcbt` instruction has one syntax form and does not affect the Condition Register field 0 or the Fixed-Point Exception register.

**Parameters**

*RA*  Specifies source general-purpose register for EA computation.

*RB*  Specifies source general-purpose register for EA computation.

*TH*  Indicates when a sequence of data cache blocks might be needed.

**Examples**

The following code sums the content of a one-dimensional vector:

```plaintext
# Assume that GPR 4 contains the address of the first element
# of the sum.
# Assume 49 elements are to be summed.
# Assume the data cache block size is 32 bytes.
# Assume the elements are word aligned and the address
# are multiples of 4.
  dcbt 0,4  # Issue hint to fetch first
target block.
  addi 5,4,32  # Compute address of second
target cache block.
  addi 8,0,6  # Set outer loop count.
  addi 7,0,8  # Set inner loop counter.
  dcbt 0,5  # Issue hint to fetch second
target cache block.
  lw 3,4,0  # Set sum = element number 1.

bigloop:
  addi 8,0,-1  # Decrement outer loop count
  and set CR field 0.
  mtsp  CTR,7  # Set counter (CTR) for
  # inner loop.
  addi 5,5,32  # Compute address for next
  # touch.

lttlloop:
  lwz 6,4,4  # Fetch element.
  add 3,3,6  # Add to sum.
  bc 16,0,lttlloop  # Decrement CTR and branch
  # if result is not equal to 0.
  dcbt 0,5  # Issue hint to fetch next
  # cache block.
```
bc 4,3,bigloop  # Branch if outer loop CTR is
        # not equal to 0.
end  # Summation complete.

Related Information
The ccls (Cache Line Compute Size) instruction, clf (Cache Line Flush) instruction, cli (Cache Line Invalidate) instruction, dcbf (Data Cache Block Flush) instruction, dcbi (Data Cache Block Invalidate) instruction, dcbst (Data Cache Block Store) instruction, dcbtst (Data Cache Block Touch for Store) instruction, dcbz or dclz (Data Cache Block Set to Zero) instruction, dclst (Data Cache Line Store) instruction, icbi (Instruction Cache Block Invalidate) instruction, sync (Synchronize) or dcs (Data Cache Synchronize) instruction.

dcbtst (Data Cache Block Touch for Store) Instruction

Purpose
Allows a program to request a cache block fetch before it is actually needed by the program.

Note: The dcbtst instruction is supported only in the PowerPC architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>246</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

PowerPC
dcbtst

RA RB

Description
The dcbtst instruction improves performance by anticipating a store to the addressed byte. The block containing the byte addressed by the effective address (EA) is fetched into the data cache before the block is needed by the program. The program can later perform stores to the block and may not experience the added delay caused by fetching the block into the cache. Executing the dcbtst instruction does not invoke the system error handler.

The dcbtst instruction calculates an effective address (EA) by adding the contents of general-purpose register (GPR) RA to the contents of GPR RB. If the RA field is 0, EA is the sum of the contents of RB and 0.

Consider the following when using the dcbtst instruction:
• If the EA specifies a direct store segment address, the instruction is treated as a no-op.
• The access is treated as a load from the addressed cache block with respect to protection. If protection does not permit access to the addressed byte, the dcbtst instruction performs no operations.
• If a program does not need to store to the data cache block, use the \texttt{dcbt} (Data Cache Block Touch) instruction.

The \texttt{dcbtst} instruction has one syntax form and does not affect Condition Register field 0 or the Fixed-Point Exception register.

\textbf{Parameters}

- \texttt{RA} Specifies source general-purpose register for operation.
- \texttt{RB} Specifies source general-purpose register for operation.

\textbf{Related Information}

The \texttt{clcs} (Cache Line Compute Size) instruction, \texttt{clf} (Cache Line Flush) instruction, \texttt{cili} (Cache Line Invalidate) instruction, \texttt{dcbf} (Data Cache Block Flush) instruction, \texttt{dcbi} (Data Cache Block Invalidate) instruction, \texttt{dcbst} (Data Cache Block Store) instruction, \texttt{dcbt} (Data Cache Block Touch) instruction, \texttt{dcbz} or \texttt{dclz} (Data Cache Block Set to Zero) instruction, \texttt{dclist} (Data Cache Line Store) instruction, \texttt{icbi} (Instruction Cache Block Invalidate) instruction, \texttt{sync} (Synchronize) or \texttt{dcs} (Data Cache Synchronize) instruction.

\textbf{Processing and Storage}

\texttt{dcbz} or \texttt{dclz} (Data Cache Block Set to Zero) Instruction

\textbf{Purpose}

The PowerPC instruction, \texttt{dcbz}, sets all bytes of a cache block to 0.

The POWER family instruction, \texttt{dclz}, sets all bytes of a cache line to 0.

\textbf{Syntax}

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>1014</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

\textbf{Description}

The \texttt{dcbz} and \texttt{dclz} instructions work with data cache blocks and data cache lines respectively. If \texttt{RA} is not 0, the \texttt{dcbz} and \texttt{dclz} instructions compute an effective address (EA) by adding the contents of general-purpose register (GPR) \texttt{RA} to the contents of GPR \texttt{RB}. If \texttt{GPR RA} is 0, the EA is the contents of GPR \texttt{RB}. 
If the cache block or line containing the addressed byte is in the data cache, all bytes in the block or line are set to 0. Otherwise, the block or line is established in the data cache without reference to storage and all bytes of the block or line are set to 0.

For the POWER family instruction \texttt{dclz}, if GPR RA is not 0, the EA replaces the content of GPR RA.

The \texttt{dcbz} and \texttt{dclz} instructions are treated as a store to the addressed cache block or line with respect to protection.

The \texttt{dcbz} and \texttt{dclz} instructions have one syntax form and do not effect the Fixed-Point Exception Register. If bit 31 is set to 1, the instruction form is invalid.

**Parameters**

\textbf{PowerPC}

- \textit{RA} Specifies the source register for EA computation.
- \textit{RB} Specifies the source register for EA computation.

\textbf{POWER family}

- \textit{RA} Specifies the source register for EA computation and the target register for EA update.
- \textit{RB} Specifies the source register for EA computation.

**Security**

The \texttt{dclz} instruction is privileged.

**Related Information**

The \texttt{clcs} (Cache Line Compute Size) instruction, \texttt{clf} (Cache Line Flush) instruction, \texttt{cli} (Cache Line Invalidate) instruction, \texttt{dcbf} (Data Cache Block Flush) instruction, \texttt{dcbi} (Data Cache Block Invalidate) instruction, \texttt{dcbst} (Data Cache Block Store) instruction, \texttt{dcbt} (Data Cache Block Touch) instruction, \texttt{dcbstt} (Data Cache Block Touch for Store) instruction, \texttt{dclst} (Data Cache Line Store) instruction, \texttt{icbi} (Instruction Cache Block Invalidate) instruction, \texttt{sync} (Synchronize) or \texttt{dcs} (Data Cache Synchronize) instruction.

**Fixed-Point Processor**

\textbf{dclst (Data Cache Line Store) Instruction}

**Purpose**

Stores a line of modified data in the data cache into main memory.

\textbf{Note:} The \texttt{dclst} instruction is supported only in the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>//</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>630</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
Description
The \texttt{dclst} instruction adds the contents of general-purpose register (GPR) \texttt{RA} to the contents of GPR \texttt{RB}. It then stores the sum in \texttt{RA} as the effective address (EA) if \texttt{RA} is not 0 and the instruction does not cause a Data Storage interrupt.

If \texttt{RA} is 0, the effective address (EA) is the sum of the contents of GPR \texttt{RB} and 0.

Consider the following when using the \texttt{dclst} instruction:

- If the line containing the byte addressed by the EA is in the data cache and has been modified, the \texttt{dclst} instruction writes the line to main memory.
- If data address translation is enabled (that is, the Machine State Register (MSR) Data Relocate (DR) bit is 1) and the virtual address has no translation, a Data Storage interrupt occurs with bit 1 of the Data Storage Interrupt Segment Register set to 1.
- If data address translation is enabled (MSR DR bit is 1), the virtual address translates to an unusable real address, the line exists in the data cache, and a Machine Check interrupt occurs.
- If data address translation is disabled (MSR DR bit is 0) the address specifies an unusable real address, the line exists in the data cache, and a Machine Check interrupt occurs.
- If the EA specifies an I/O address, the instruction is treated as a no-op, but the effective address is placed into GPR \texttt{RA}.
- Address translation treats the \texttt{dclst} instruction as a load to the byte addressed, ignoring protection and data locking. If this instruction causes a Translation Look-Aside Buffer (TLB) miss, the reference bit is set.

The \texttt{dclst} instruction has one syntax form and does not effect the Fixed-Point Exception register. If the Record (Rc) bit is set to 1, Condition Register Field 0 is undefined.

Parameters

\texttt{RA} \hspace{1em} Specifies the source and target general-purpose register where result of operation is stored.

\texttt{RB} \hspace{1em} Specifies the source general-purpose register for EA calculation.

Examples

The following code stores the sum of the contents of GPR 4 and GPR 6 in GPR 6 as the effective address:

```
# Assume that GPR 4 contains 0x0000 3000.
# Assume that GPR 6 is the target register and that it
# contains 0x0000 0000.
dclst 6,4
# GPR 6 now contains 0x0000 3000.
```

Related Information

The \texttt{clcs} (Cache Line Compute Size) instruction, \texttt{clf} (Cache Line Flush) instruction, \texttt{cli} (Cache Line Invalidate) instruction, \texttt{dcbf} (Data Cache Block Flush) instruction, \texttt{dcbi} (Data Cache Block Invalidate) instruction, \texttt{dcbst} (Data Cache Block Store) instruction, \texttt{dcbt} (Data Cache Block Touch) instruction, \texttt{dcbstt} (Data Cache Block Touch for Store) instruction, \texttt{dcbz} or \texttt{dclz} (Data Cache Block Set to Zero) instruction, \texttt{icbi} (Instruction Cache Block Invalidate) instruction, \texttt{sync} (Synchronize) or \texttt{dcs} (Data Cache Synchronize) instruction.
**div (Divide) Instruction**

**Purpose**
Divides the contents of a general-purpose register concatenated with the MQ Register by the contents of a general-purpose register and stores the result in a general-purpose register.

**Note:** The div instruction is supported only in the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>331</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**POWER family**
- div: $RT RA RB$
- div.: $RT RA RB$
- divo: $RT RA RB$
- divo.: $RT RA RB$

**Description**
The div instruction concatenates the contents of general-purpose register (GPR) $RA$ and the contents of Multiply Quotient (MQ) Register, divides the result by the contents of GPR $RB$, and stores the result in the target GPR $RT$. The remainder has the same sign as the dividend, except that a zero quotient or a zero remainder is always positive. The results obey the equation:

$$\text{dividend} = (\text{divisor} \times \text{quotient}) + \text{remainder}$$

where a dividend is the original $(RA) \parallel (MQ)$, divisor is the original $(RB)$, quotient is the final $(RT)$, and remainder is the final $(MQ)$.

For the case of $-2^{31} \leq -1$, the MQ Register is set to 0 and $-2^{31}$ is placed in GPR $RT$. For all other overflows, the contents of MQ, the target GPR $RT$, and the Condition Register Field 0 (if the Record Bit (Rc) is 1) are undefined.

The div instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>div</td>
<td>0</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>div.</td>
<td>0</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>divo</td>
<td>1</td>
<td>SO,OV</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>divo.</td>
<td>1</td>
<td>SO,OV</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>
The four syntax forms of the `div` instruction never affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- **RT**  Specifies target general-purpose register where result of operation is stored.
- **RA**  Specifies source general-purpose register for operation.
- **RB**  Specifies source general-purpose register for operation.

**Examples**

1. The following code divides the contents of GPR 4, concatenated with the MQ Register, by the contents of GPR 6 and stores the result in GPR 4:

```plaintext
# Assume the MQ Register contains 0x0000 0001.
# Assume GPR 4 contains 0x0000 0000.
# Assume GPR 6 contains 0x0000 0002.
div 4,4,6
# GPR 4 now contains 0x0000 0000.
# The MQ Register now contains 0x0000 0001.
```

2. The following code divides the contents of GPR 4, concatenated with the MQ Register, by the contents of GPR 6, stores the result in GPR 4, and sets Condition Register Field 0 to reflect the result of the operation:

```plaintext
# Assume the MQ Register contains 0x0000 0002.
# Assume GPR 4 contains 0x0000 0000.
# Assume GPR 6 contains 0x0000 0002.
div 4,4,6
# GPR 4 now contains 0x0000 0001.
# MQ Register contains 0x0000 0000.
```

3. The following code divides the contents of GPR 4, concatenated with the MQ Register, by the contents of GPR 6, places the result in GPR 4, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register to reflect the result of the operation:

```plaintext
# Assume GPR 4 contains 0x0000 0001.
# Assume GPR 6 contains 0x0000 0000.
# Assume the MQ Register contains 0x0000 0000.
divo 4,4,6
# GPR 4 now contains an undefined quantity.
# The MQ Register is undefined.
```

4. The following code divides the contents of GPR 4, concatenated with the MQ Register, by the contents of GPR 6, places the result in GPR 4, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```plaintext
# Assume GPR 4 contains 0x-1.
# Assume GPR 6 contains 0x2.
# Assume the MQ Register contains 0xFFFFFFFF.
divo 4,4,6
# GPR 4 now contains 0x0000 0000.
# The MQ Register contains 0x-1.
```

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Arithmetic Instructions](#)
**divd (Divide Double Word) Instruction**

**Purpose**
Divide the contents of a general purpose register by the contents of a general purpose register, storing the result into a general purpose register.

This instruction should only be used on 64-bit PowerPC® processors running a 64-bit application.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>489</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC64
- **divd**  \( RT, RA, RB \) (OE=0 Rc=0)
- **divd.**  \( RT, RA, RB \) (OE=0 Rc=1)
- **divdo**  \( RT, RA, RB \) (OE=1 Rc=0)
- **divdo.**  \( RT, RA, RB \) (OE=1 Rc=1)

**Description**
The 64-bit dividend is the contents of \( RA \). The 64-bit divisor is the contents of \( RB \). The 64-bit quotient is placed into \( RT \). The remainder is not supplied as a result.

Both the operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies the equation: \( \text{dividend} = (\text{quotient} \times \text{divisor}) + r \), where \( 0 \leq r < \text{divisor} \) if the dividend is non-negative, and \( -\text{divisor} < r \leq 0 \) if the dividend is negative.

If an attempt is made to perform the divisions 0x8000_0000_0000_0000 / -1 or / 0, the contents of \( RT \) are undefined, as are the contents of the LT, GT, and EQ bits of the condition register 0 field (if the record bit \( \text{Rc} \) = 1 (the **divd.** or **divdo.** instructions)). In this case, if overflow enable (OE) = 1 then the overflow bit (OV) is set.

The 64-bit signed remainder of dividing \( RA \) by \( RB \) can be computed as follows, except in the case that \( RA = -2^{63} \) and \( RB = -1 \):

- **divd**  \( RT, RA, RB \)  
  \[ # \ RT = \text{quotient} \]
- **mulld**  \( RT, RT, RB \)  
  \[ # \ RT = \text{quotient} \times \text{divisor} \]
- **subf**  \( RT, RT, RA \)  
  \[ # \ RT = \text{remainder} \]

**Parameters**
- \( RT \)  
  Specifies target general-purpose register for the result of the computation.
- \( RA \)  
  Specifies source general-purpose register for the dividend.
Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

divdu (Divide Double Word Unsigned) Instruction

Purpose
Divide the contents of a general purpose register by the contents of a general purpose register, storing the result into a general purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>457</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC

- divdu: \( RT, RA, RB \) (OE=0 Rc=0)
- divdu: \( RT, RA, RB \) (OE=0 Rc=1)
- divduo: \( RT, RA, RB \) (OE=1 Rc=0)
- divduo: \( RT, RA, RB \) (OE=1 Rc=1)

Description
The 64-bit dividend is the contents of \( RA \). The 64-bit divisor is the contents of \( RB \). The 64-bit quotient is placed into \( RT \). The remainder is not supplied as a result.

Both the operands and the quotient are interpreted as unsigned integers, except that if the record bit (Rc) is set to 1 the first three bits of the condition register 0 (CR0) field are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies the equation: dividend = (quotient \* divisor) + r, where 0 <= r < divisor.

If an attempt is made to perform the division \( \text{anything} \) / 0 the contents of \( RT \) are undefined, as are the contents of the LT, GT, and EQ bits of the CR0 field (if Rc = 1). In this case, if the overflow enable bit (OE) = 1 then the overflow bit (OV) is set.

The 64-bit unsigned remainder of dividing \( RA \) by \( RB \) can be computed as follows:

- \text{divdu} \quad RT, RA, RB  # RT = quotient
- \text{mulld} \quad RT, RT, RB  # RT = quotient \* divisor
- \text{subf} \quad RT, RT, RA  # RT = remainder
Other registers altered:

• Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
• XER: Affected: SO, OV (if OE = 1)

**Note:** The setting of the affected bits in the XER is mode-independent, and reflects overflow of the 64-bit result.

**Parameters**

<table>
<thead>
<tr>
<th>RT</th>
<th>Specifies target general-purpose register for the result of the computation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>Specifies source general-purpose register for the dividend.</td>
</tr>
<tr>
<td>RB</td>
<td>Specifies source general-purpose register for the divisor.</td>
</tr>
</tbody>
</table>

**Implementation**

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

**divs (Divide Short) Instruction**

**Purpose**

Divides the contents of a general-purpose register by the contents of a general-purpose register and stores the result in a general-purpose register.

**Note:** The **divs** instruction is supported only in the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22-30</td>
<td>363</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**Description**

The **divs** instruction divides the contents of general-purpose register (GPR) **RA** by the contents of GPR **RB** and stores the result in the target GPR **RT**. The remainder has the same sign as the dividend, except that a zero quotient or a zero remainder is always positive. The results obey the equation:

\[
\text{dividend} = (\text{divisor} \times \text{quotient}) + \text{remainder}
\]
where a dividend is the original \((RA)\), divisor is the original \((RB)\), quotient is the final \((RT)\), and remainder is the final \((MQ)\).

For the case of \(-2^{*31} P -1\), the MQ Register is set to 0 and \(-2^{*31}\) is placed in GPR \(RT\). For all other overflows, the contents of MQ, the target GPR \(RT\) and the Condition Register Field 0 (if the Record Bit (Rc) is 1) are undefined.

The \texttt{divs} instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>divs</td>
<td>0</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>divs.</td>
<td>0</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>divso</td>
<td>1</td>
<td>SO,OV</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>divso.</td>
<td>1</td>
<td>SO,OV</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The four syntax forms of the \texttt{divs} instruction never affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- \(RT\) Specifies target general-purpose register where result of operation is stored.
- \(RA\) Specifies source general-purpose register for operation.
- \(RB\) Specifies source general-purpose register for operation.

**Examples**

1. The following code divides the contents of GPR 4 by the contents of GPR 6 and stores the result in GPR 4:
   
   ```
   # Assume GPR 4 contains 0x0000 0001.
   # Assume GPR 6 contains 0x0000 0002.
   divs 4,4,6
   # GPR 4 now contains 0x0.
   # The MQ Register now contains 0x1.
   ```

2. The following code divides the contents of GPR 4 by the contents of GPR 6, stores the result in GPR 4 and sets Condition Register Field 0 to reflect the result of the operation:
   
   ```
   # Assume GPR 4 contains 0x0000 0002.
   # Assume GPR 6 contains 0x0000 0002.
   divs. 4,4,6
   # GPR 4 now contains 0x0000 0001.
   # The MQ Register now contains 0x0000 0000.
   ```

3. The following code divides the contents of GPR 4 by the contents of GPR 6, stores the result in GPR 4, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register to reflect the result of the operation:
   
   ```
   # Assume GPR 4 contains 0x0000 0001.
   # Assume GPR 6 contains 0x0000 0000.
   divso 4,4,6
   # GPR 4 now contains an undefined quantity.
   ```
4. The following code divides the contents of GPR 4 by the contents of GPR 6, stores the result in GPR 4, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

Assume GPR 4 contains 0x-1.
Assume GPR 6 contains 0x0000 00002.
Assume the MQ Register contains 0x0000 0000.
divso. 4,4,6
GPR 4 now contains 0x0000 0000.
The MQ register contains 0x-1.

Related Information

- Fixed-Point Processor
- Fixed-Point Arithmetic Instructions

**divw (Divide Word) Instruction**

**Purpose**

Divides the contents of a general-purpose register by the contents of another general-purpose register and stores the result in a third general-purpose register.

*Note:* The divw instruction is supported only in the PowerPC architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>491</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**PowerPC**

- divw  
  - RT | RA | RB
- divw. 
  - RT | RA | RB
- divwo 
  - RT | RA | RB
- divwo. 
  - RT | RA | RB

**Description**

The divw instruction divides the contents of general-purpose register (GPR) RA by the contents of GPR RB, and stores the result in the target GPR RT. The dividend, divisor, and quotient are interpreted as signed integers.

For the case of \(-2^{31} / -1\), and all other cases that cause overflow, the content of GPR RT is undefined.

The divw instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.
The four syntax forms of the `divw` instruction never affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

### Parameters

- **RT** Specifies target general-purpose register where result of operation is stored.
- **RA** Specifies source general-purpose register for dividend.
- **RB** Specifies source general-purpose register for divisor.

### Examples

1. The following code divides the contents of GPR 4 by the contents of GPR 6 and stores the result in GPR 4:

   ```
   # Assume GPR 4 contains 0x0000 0000.
   # Assume GPR 6 contains 0x0000 0002.
   divw 4,4,6
   # GPR 4 now contains 0x0000 0000.
   ```

2. The following code divides the contents of GPR 4 by the contents of GPR 6, stores the result in GPR 4 and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x0000 0002.
   # Assume GPR 6 contains 0x0000 0002.
   divw. 4,4,6
   # GPR 4 now contains 0x0000 0001.
   ```

3. The following code divides the contents of GPR 4 by the contents of GPR 6, places the result in GPR 4, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x0000 0001.
   # Assume GPR 6 contains 0x0000 0000.
   divwo 4,4,6
   # GPR 4 now contains an undefined quantity.
   ```

4. The following code divides the contents of GPR 4 by the contents of GPR 6, places the result in GPR 4, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x0000 0000.
   # Assume GPR 6 contains 0xFFFF FFFF.
   divwo. 4,4,6
   # GPR 4 now contains undefined quantity.
   ```

### Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Arithmetic Instructions](#)
divwu (Divide Word Unsigned) Instruction

Purpose
Divides the contents of a general-purpose register by the contents of another general-purpose register and stores the result in a third general-purpose register.

Note: The divwu instruction is supported only in the PowerPC architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>459</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC

\[
\text{divwu } RT, RA, RB \\
\text{divwu. } RT, RA, RB \\
\text{divwuo } RT, RA, RB \\
\text{divwuo. } RT, RA, RB
\]

Description
The divwu instruction divides the contents of general-purpose register (GPR) RA by the contents of GPR RB, and stores the result in the target GPR RT. The dividend, divisor, and quotient are interpreted as unsigned integers.

For the case of division by 0, the content of GPR RT is undefined.

Note: Although the operation treats the result as an unsigned integer, if Rc is 1, the Less Than (LT) zero, Greater Than (GT) zero, and Equal To (EQ) zero bits of Condition Register Field 0 are set as if the result were interpreted as a signed integer.

The divwu instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>divwu</td>
<td>0</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>divwu.</td>
<td>0</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>divwuo</td>
<td>1</td>
<td>SO, OV,</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>divwuo.</td>
<td>1</td>
<td>SO, OV,</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The four syntax forms of the divwu instruction never affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary
Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- **RT** Specifies target general-purpose register where result of operation is stored.
- **RA** Specifies source general-purpose register for EA calculation.
- **RB** Specifies source general-purpose register for EA calculation.

**Examples**

1. The following code divides the contents of GPR 4 by the contents of GPR 6 and stores the result in GPR 4:

   ```
   # Assume GPR 4 contains 0x0000 0000.
   # Assume GPR 6 contains 0x0000 0002.
   divwu 4,4,6
   # GPR 4 now contains 0x0000 0000.
   ```

2. The following code divides the contents of GPR 4 by the contents of GPR 6, stores the result in GPR 4 and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x0000 0002.
   # Assume GPR 6 contains 0x0000 0002.
   divw 4,4,6
   # GPR 4 now contains 0x0000 0001.
   ```

3. The following code divides the contents of GPR 4 by the contents of GPR 6, places the result in GPR 4, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x0000 0001.
   # Assume GPR 6 contains 0x0000 0000.
   divwu 4,4,6
   # GPR 4 now contains 0x0000 0000.
   ```

4. The following code divides the contents of GPR 4 by the contents of GPR 6, places the result in GPR 4, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x8000 0000.
   # Assume GPR 6 contains 0x0000 0002.
   divwuo 4,4,6
   # GPR 4 now contains 0x4000 0000.
   ```

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Arithmetic Instructions](#)

---

**doz (Difference or Zero) Instruction**

**Purpose**

Computes the difference between the contents of two general-purpose registers and stores the result or the value zero in a general-purpose register.

**Note:** The doz instruction is supported only in the POWER family architecture.
**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>264</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**Description**

The `doz` instruction adds the complement of the contents of general-purpose register (GPR) `RA`, 1, and the contents of GPR `RB`, and stores the result in the target GPR `RT`.

If the value in GPR `RA` is algebraically greater than the value in GPR `RB`, then GPR `RT` is set to 0.

The `doz` instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>doz</td>
<td>0</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>doz.</td>
<td>0</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EO,SO</td>
</tr>
<tr>
<td>dozo</td>
<td>1</td>
<td>SO,OV</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>dozo.</td>
<td>1</td>
<td>SO,OV</td>
<td>1</td>
<td>LT,GT,EO,SO</td>
</tr>
</tbody>
</table>

The four syntax forms of the `doz` instruction never affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register; the Overflow (OV) bit can only be set on positive overflows. If the syntax form sets the Record (Rc) bit to 1, the instruction effects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- `RT` Specifies target general-purpose register where result of operation is stored.
- `RA` Specifies source general-purpose register for operation.
- `RB` Specifies source general-purpose register for operation.
Examples

1. The following code determines the difference between the contents of GPR 4 and GPR 6 and stores the result in GPR 4:

```plaintext
# Assume GPR 4 holds 0x0000 0001.
# Assume GPR 6 holds 0x0000 0002.
doz 4,4,6
# GPR 4 now holds 0x0000 0001.
```

2. The following code determines the difference between the contents of GPR 4 and GPR 6, stores the result in GPR 4, and sets Condition Register Field 0 to reflect the result of the operation:

```plaintext
# Assume GPR 4 holds 0x0000 0001.
# Assume GPR 6 holds 0x0000 0000.
doz 4,4,6
# GPR 4 now holds 0x0000 0000.
```

3. The following code determines the difference between the contents of GPR 4 and GPR 6, stores the result in GPR 4, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register to reflect the result of the operation:

```plaintext
# Assume GPR 4 holds 0x0000 0002.
# Assume GPR 6 holds 0x0000 0000.
dozo 4,4,6
# GPR 4 now holds 0x0000 0006.
```

4. The following code determines the difference between the contents of GPR 4 and GPR 6, stores the result in GPR 4, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```plaintext
# Assume GPR 4 holds 0xEFFF FFFF.
# Assume GPR 6 holds 0x0000 0000.
dozo 4,4,6
# GPR 4 now holds 0x1000 0001.
```

Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Arithmetic Instructions](#)

**dozi (Difference or Zero Immediate) Instruction**

**Purpose**

Computes the difference between the contents of a general-purpose register and a signed 16-bit integer and stores the result or the value zero in a general-purpose register.

**Note:** The `dozi` instruction is supported only in the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>09</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>SI</td>
</tr>
</tbody>
</table>

POWER family

`dozi` [RT RA SI]
Description

The `dozi` instruction adds the complement of the contents of general-purpose register (GPR) `RA`, the 16-bit signed integer `SI`, and 1 and stores the result in the target GPR `RT`.

If the value in GPR `RA` is algebraically greater than the 16-bit signed value in the `SI` field, then GPR `RT` is set to 0.

The `dozi` instruction has one syntax form and does not effect Condition Register Field 0 or the Fixed-Point Exception Register.

Parameters

- **RT**: Specifies target general-purpose register where result of operation is stored.
- **RA**: Specifies source general-purpose register for operation.
- **SI**: Specifies signed 16-bit integer for operation.

Examples

The following code determines the difference between GPR 4 and 0x0 and stores the result in GPR 4:

```assembly
# Assume GPR 4 holds 0x0000 0001.
dozi 4,4,0x0
# GPR 4 now holds 0x0000 0000.
```

Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Arithmetic Instructions](#)

eciwx (External Control In Word Indexed) Instruction

Purpose

Translates the effective address (EA) to a real address, sends the real address to a controller, and loads the word returned by the controller into a register.

Note: The `eciwx` instruction is defined only in the PowerPC architecture and is an optional instruction. It is supported on the PowerPC 601 RISC Microprocessor, PowerPC 603 RISC Microprocessor, and PowerPC 604 RISC Microprocessor.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>310</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

eciwx  

Assembler Language Reference
Description
The `eciwx` instruction translates EA to a real address, sends the real address to a controller, and places the word returned by the controller in general-purpose register `RT`. If `RA = 0`, the EA is the content of `RB`, otherwise EA is the sum of the content of `RA` plus the content of `RB`.

If EAR(E) = 1, a load request for the real address corresponding to EA is sent to the controller identified by EAR(RID), bypassing the cache. The word returned by the controller is placed in `RT`.

Notes:
1. EA must be a multiple of 4 (a word-aligned address); otherwise, the result is boundedly undefined.
2. The operation is treated as a load to the addressed byte with respect to protection.

Parameters
- `RT` Specifies target general-purpose register where result of operation is stored.
- `RA` Specifies source general-purpose register for operation.
- `RB` Specifies source general-purpose register for operation.

Related Information
- "ecowx (External Control Out Word Indexed) Instruction."
  Chapter 2, “Processing and Storage,” on page 11.

ecowx (External Control Out Word Indexed) Instruction

Purpose
Translates the effective address (EA) to a real address and sends the real address and the contents of a register to a controller.

Note: The `ecowx` instruction is defined only in the PowerPC architecture and is an optional instruction. It is supported on the PowerPC 601 RISC Microprocessor, PowerPC 603 RISC Microprocessor, and PowerPC 604 RISC Microprocessor.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>438</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

```
ecowx  RS RA RB
```

Description
The `ecowx` instruction translates EA to a real address and sends the real address and the content of general-purpose register `RS` to a controller. If `RA = 0`, the EA is the content of `RB`, otherwise EA is the sum of the content of `RA` plus the content of `RB`. 
If EAR(E) = 1, a store request for the real address corresponding to EA is sent to the controller identified by EAR(RID), bypassing the cache. The content of RS is sent with the store request.

Notes:
1. EA must be a multiple of 4 (a word-aligned address); otherwise, the result is boundedly undefined.
2. The operation is treated as a store to the addressed byte with respect to protection.

Parameters

- RS: Specifies target general-purpose register where result of operation is stored.
- RA: Specifies source general-purpose register for operation.
- RB: Specifies source general-purpose register for operation.

Related Information

The **eciwx** (External Control In Word Indexed) instruction.

Processing and Storage

**eieio (Enforce In-Order Execution of I/O) Instruction**

Purpose

Ensures that cache-inhibited storage accesses are performed in main memory in the order specified by the program.

Note: The **eieio** instruction is supported only in the PowerPC architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>854</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

PowerPC

**eieio**

Description

The **eieio** instruction provides an ordering function that ensures that all load and store instructions initiated prior to the **eieio** instruction complete in main memory before any loads or stores subsequent to the **eieio** instruction access memory. If the **eieio** instruction is omitted from a program, and the memory locations are unique, the accesses to main storage may be performed in any order.
**Note:** The `eieio` instruction is appropriate for cases where the only requirement is to control the order of storage references as seen by I/O devices. However, the `sync` (Synchronize) instruction provides an ordering function for all instructions.

The `eieio` instruction has one syntax form and does not affect Condition Register Field 0 or the Fixed-Point Exception Register.

**Examples**

The following code ensures that, if the memory locations are in cache-inhibited storage, the load from location `AA` and the store to location `BB` are completed in main storage before the content of location `CC` is fetched or the content of location `DD` is updated:

```assembly
lwz r4,AA(r1)
stw r4,BB(r1)
eieio
lwz r5,CC(r1)
stw r5,DD(r1)
```

**Note:** If the memory locations of `AA`, `BB`, `CC`, and `DD` are not in cache-inhibited memory, the `eieio` instruction has no effect on the order that instructions access memory.

**Related Information**

The `sync` (Synchronize) or `dcs` (Data Cache Synchronize) instruction.

---

**extsw (Extend Sign Word) Instruction**

**Purpose**

Copy the low-order 32 bits of a general purpose register into another general purpose register, and sign extend the fullword to a double-word in size (64 bits).

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>00000</td>
</tr>
<tr>
<td>21-30</td>
<td>986</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**PowerPC**

- `extsw RA RS (Rc=0)`
- `extsw RA RS (Rc=1)`

**Description**

The contents of the low-order 32 bits of general purpose register (GPR) `RS` are placed into the low-order 32 bits of GPR `RA`. Bit 32 of GPR `RS` is used to fill the high-order 32 bits of GPR `RA`.

Other registers altered:
• Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
• XER:
  Affected: CA

Parameters

RA Specifies the target general purpose register for the result of the operation.
RS Specifies the source general purpose register for the operand of instruction.

Implementation

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

eqv (Equivalent) Instruction

Purpose

Logically XORs the contents of two general-purpose registers and places the complemented result in a general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>284</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

eqv  

eqv.

Description

The eqv instruction logically XORs the contents of general-purpose register (GPR) RS with the contents of GPR RB and stores the complemented result in the target GPR RA.

The eqv instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>eqv</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>eqv.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EO,SO</td>
</tr>
</tbody>
</table>
The two syntax forms of the `eqv` instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- **RA** Specifies target general-purpose register where result of operation is stored.
- **RS** Specifies source general-purpose register for operation.
- **RB** Specifies source general-purpose register for operation.

**Examples**

1. The following code logically XORs the contents of GPR 4 and GPR 6 and stores the complemented result in GPR 4:

   ```
   # Assume GPR 4 holds 0xFFF2 5730.
   # Assume GPR 6 holds 0x7B41 92C0.
   eqv 4,4,6
   # GPR 4 now holds 0x7B4C 3A0F.
   ```

2. The following code XORs the contents of GPR 4 and GPR 6, stores the complemented result in GPR 4, and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 holds 0x0000 00FD.
   # Assume GPR 6 holds 0x7B41 92C0.
   eqv 4,4,6
   # GPR 4 now holds 0x84BE 6DC2.
   ```

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Logical Instructions](#)

---

### extsb (Extend Sign Byte) Instruction

**Purpose**

Extends the sign of the low-order byte.

**Note:** The `extsb` instruction is supported only in the PowerPC architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>954</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**PowerPC**

- `extsb` [RA RS]
- `extsb.` [RA RS]
Description
The **extsb** instruction places bits 24-31 of general-purpose register (GPR) \( RS \) into bits 24-31 of GPR \( RA \) and copies bit 24 of register \( RS \) in bits 0-23 of register \( RA \).

The **extsb** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

- \( RA \) Specifies target general-purpose register where result of operation is stored.
- \( RS \) Specifies source general-purpose register of containing the byte to be extended.

Examples

1. The following code extends the sign of the least significant byte contained in GPR 4 and places the result in GPR 6:
   
   ```
   # Assume GPR 6 holds 0x5A5A 5A5A.
   extsb 4,6
   # GPR 6 now holds 0x0000 005A.
   ```

2. The following code extends the sign of the least significant byte contained in GPR 4 and sets Condition Register Field 0 to reflect the result of the operation:
   
   ```
   # Assume GPR 4 holds 0xA5A5 A5A5.
   extsb 4,4
   # GPR 4 now holds 0xFFFF FFA5.
   ```

Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Logical Instructions](#)

**extsh or exts (Extend Sign Halfword) Instruction**

**Purpose**
Extends the lower 16-bit contents of a general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>922</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
Description

The **extsh** and **exts** instructions place bits 16-31 of general-purpose register (GPR) RS into bits 16-31 of GPR RA and copy bit 16 of GPR RS in bits 0-15 of GPR RA.

The **extsh** and **exts** instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>extsh</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>extsh.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>exts</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>exts.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the **extsh** instruction, and the two syntax forms of the **exts** instruction, never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

**RA**  Specifies general-purpose register receives extended integer.

**RS**  Specifies source general-purpose register for operation.

Examples

1. The following code places bits 16-31 of GPR 6 into bits 16-31 of GPR 4 and copies bit 16 of GPR 6 into bits 0-15 of GPR 4:
   # Assume GPR 6 holds 0x0000 FFFF.
   `extsh 4,6`
   # GPR 6 now holds 0xFFF FFFF.

2. The following code places bits 16-31 of GPR 6 into bits 16-31 of GPR 4, copies bit 16 of GPR 6 into bits 0-15 of GPR 4, and sets Condition Register Field 0 to reflect the result of the operation:
   # Assume GPR 4 holds 0x0000 2FFF.
   `extsh 6,4`
   # GPR 6 now holds 0x0000 2FFF.

Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Logical Instructions](#)
fabs (Floating Absolute Value) Instruction

Purpose
Stores the absolute value of the contents of a floating-point register in another floating-point register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-30</td>
<td>264</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

fabs
fabs.

Description
The `fabs` instruction sets bit 0 of floating-point register (FPR) `FRB` to 0 and places the result into FPR `FRT`.

The `fabs` instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fabs</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fabs.</td>
<td>None</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

The two syntax forms of the `fabs` instruction never affect the Floating-Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception Summary (FX), Floating-Point Enabled Exception Summary (FEX), Floating-Point Invalid Operation Exception Summary (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

Parameters

- `FRT` Specifies target floating-point register for operation.
- `FRB` Specifies source floating-point register for operation.

Examples

1. The following code sets bit 0 of FPR 4 to zero and places the result in FPR 6:
   
   ```
   # Assume FPR 4 holds 0xC053 4000 0000 0000.
   fabs 6,4
   # GPR 6 now holds 0x4053 4000 0000 0000.
   ```

2. The following code sets bit 0 of FPR 25 to zero, places the result in FPR 6, and sets Condition Register Field 1 to reflect the result of the operation:
# Assume FPR 25 holds 0xFFFF FFFF FFFF FFFF.
fabs. 6,25
# GPR 6 now holds 0x7FFF FFFF FFFF FFFF.

Related Information

- Floating-Point Processor
- Floating-Point Move Instructions
- Interpreting the Contents of a Floating-Point Register

fadd or fa (Floating Add) Instruction

Purpose
Adds two floating-point operands and places the result in a floating-point register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-25</td>
<td>///</td>
</tr>
<tr>
<td>26-30</td>
<td>21</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC

- fadd
  - FRT FRA FRB
- fadd.
  - FRT FRA FRB

POWER family

- fa
  - FRT FRA FRB
- fa.
  - FRT FRA FRB

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>59</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-25</td>
<td>///</td>
</tr>
<tr>
<td>26-30</td>
<td>21</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC

- fadds
  - FRT FRA FRB
- fadds.
  - FRT FRA FRB
Description

The **fadd** and **fa** instructions add the 64-bit, double-precision floating-point operand in floating-point register (FPR) **FRA** to the 64-bit, double-precision floating-point operand in FPR **FRB**.

The **fadds** instruction adds the 32-bit single-precision floating-point operand in FPR **FRA** to the 32-bit single-precision floating-point operand in FPR **FRB**.

The result is rounded under control of the Floating-Point Rounding Control Field **RN** of the Floating-Point Status and Control Register and is placed in FPR **FRT**.

Addition of two floating-point numbers is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added algebraically to form the intermediate sum. All 53 bits in the significand as well as all three guard bits (G, R and X) enter into the computation.

The Floating-Point Result Field of the Floating-Point Status and Control Register is set to the class and sign of the result except for Invalid Operation exceptions when the Floating-Point Invalid Operation Exception Enable (VE) bit of the Floating-Point Status and Control Register is set to 1.

The **fadd**, **fadds**, and **fa** instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fadd</strong></td>
<td>C,FL,FG,FE,FU,FI,OX,UX,XX,VXSNAN,VXSI</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><strong>fadd.</strong></td>
<td>C,FL,FG,FE,FU,FI,OX,UX,XX,VXSNAN,VXSI</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
<tr>
<td><strong>fadds</strong></td>
<td>C,FL,FG,FE,FU,FI,OX,UX,XX,VXSNAN,VXSI</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><strong>fadds.</strong></td>
<td>C,FL,FG,FE,FU,FI,OX,UX,XX,VXSNAN,VXSI</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
<tr>
<td><strong>fa</strong></td>
<td>C,FL,FG,FE,FU,FI,OX,UX,XX,VXSNAN,VXSI</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><strong>fa.</strong></td>
<td>C,FL,FG,FE,FU,FI,OX,UX,XX,VXSNAN,VXSI</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

All syntax forms of the **fadd**, **fadds**, and **fa** instructions always affect the Floating-Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception Summary (FX), Floating-Point Enabled Exception Summary (FEX), Floating-Point Invalid Operation Exception Summary (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

Parameters

- **FRT** Specifies target floating-point register for operation.
- **FRA** Specifies source floating-point register for operation.
- **FRB** Specifies source floating-point register for operation.

Examples

1. The following code adds the contents of FPR 4 and FPR 5, places the result in FPR 6, and sets the Floating-Point Status and Control Register to reflect the result of the operation:
Assume FPR 4 contains 0xC053 4000 0000 0000.
Assume FPR 5 contains 0x400C 0000 0000 0000.
fadd 6,4,5
# FPR 6 now contains 0xC052 6000 0000 0000.

2. The following code adds the contents of FPR 4 and FPR 25, places the result in FPR 6, and sets Condition Register Field 1 and the Floating-Point Status and Control Register to reflect the result of the operation:

Assume FPR 4 contains 0xC053 4000 0000 0000.
Assume FPR 25 contains 0xFFFF FFFF FFFF FFFF.
fadd 6,4,25
# GPR 6 now contains 0xFFFF FFFF FFFF FFFF.

Related Information
Floating-Point Processor.
Floating-Point Arithmetic Instructions.
Interpreting the Contents of a Floating-Point Register.

fcfid (Floating Convert from Integer Double Word) Instruction

Purpose
Convert the fixed-point contents of a floating-point register to a double-precision floating-point number.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>00000</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-30</td>
<td>846</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC
fcfid  FRT FRB \( (Rc=0) \)
fcfid. FRT FRB \( (Rc=1) \)

Description
The 64-bit signed fixed-point operand in floating-point register (FPR) \( FRB \) is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision using the rounding mode specified by FPSCR[RN] and placed into FPR \( FRT \).

FPSCR[FPRF] is set to the class and sign of the result. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

The fcfid instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parameters

\( FRT \) Specifies the target floating-point register for the operation.

\( FRB \) Specifies the source floating-point register for the operation.

Implementation

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

**fcmpo (Floating Compare Ordered) Instruction**

**Purpose**

Compares the contents of two floating-point registers.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-8</td>
<td>BF</td>
</tr>
<tr>
<td>9-10</td>
<td>//</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-30</td>
<td>32</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

\[ \text{fcmpo BF, FRA, FRB} \]

**Description**

The `fcmpo` instruction compares the 64-bit, double-precision floating-point operand in floating-point register (FPR) \( FRA \) to the 64-bit, double-precision floating-point operand in FPR \( FRB \). The Floating-Point Condition Code Field (FPCC) of the Floating-Point Status and Control Register (FPSCR) is set to reflect the value of the operand FPR \( FRA \) with respect to operand FPR \( FRB \). The value \( BF \) determines which field in the condition register receives the four FPCC bits.

Consider the following when using the `fcmpo` instruction:

- If one of the operands is either a Quiet NaN (QNaN) or a Signaling NaN (SNaN), the Floating-Point Condition Code is set to reflect unordered (FU).
- If one of the operands is a SNaN, then the Floating-Point Invalid Operation Exception bit VXSNAN of the Floating-Point Status and Control Register is set. Also:
  - If Invalid Operation is disabled (that is, the Floating-Point Invalid Operation Exception Enable bit of the Floating-Point Status and Control Register is 0), then the Floating-Point Invalid Operation Exception bit VXVC is set (signaling an an invalid compare).
If one of the operands is a QNaN, then the Floating-Point Invalid Operation Exception bit VXVC is set.

The fcmpo instruction has one syntax form and always affects the FT, FG, FE, FU, VXSNAN, and VXVC bits in the Floating-Point Status and Control Register.

Parameters

- **BF** Specifies field in the condition register that receives the four FPCC bits.
- **FRA** Specifies source floating-point register.
- **FRB** Specifies source floating-point register.

Examples

The following code compares the contents of FPR 4 and FPR 6 and sets Condition Register Field 1 and the Floating-Point Status and Control Register to reflect the result of the operation:

```
# Assume CR = 0 and FPSCR = 0.
# Assume FPR 5 contains 0xC053 4000 0000 0000.
# Assume FPR 4 contains 0x400C 0000 0000 0000.
# fcmpo 6,4,5
# CR now contains 0x0000 0040.
# FPSCR now contains 0x0000 4000.
```

Related Information

- Floating-Point Processor
- Floating-Point Compare Instructions

**fcmpu (Floating Compare Unordered) Instruction**

**Purpose**

Compares the contents of two floating-point registers.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-8</td>
<td>BF</td>
</tr>
<tr>
<td>9-10</td>
<td>//</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-30</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

```
fcmpu BF FRA FRB
```

**Description**

The fcmpu instruction compares the 64-bit double precision floating-point operand in floating-point register (FPR) **FRA** to the 64-bit double precision floating-point operand in FPR **FRB**. The Floating-Point Condition
Code Field (FPCC) of the Floating-Point Status and Control Register (FPSCR) is set to reflect the value of the operand FRA with respect to operand FRB. The value BF determines which field in the condition register receives the four FPCC bits.

Consider the following when using the `fcmpu` instruction:

- If one of the operands is either a Quiet NaN or a Signaling NaN, the Floating-Point Condition Code is set to reflect unordered (FU).
- If one of the operands is a Signaling NaN, then the Floating-Point Invalid Operation Exception bit VXSNAN of the Floating-Point Status and Control Register is set.

The `fcmpu` instruction has one syntax form and always affects the FT, FG, FE, FU, and VXSNAN bits in the FPSCR.

**Parameters**

- BF  Specifies a field in the condition register that receives the four FPCC bits.
- FRA  Specifies source floating-point register.
- FRB  Specifies source floating-point register.

**Examples**

The following code compares the contents of FPR 5 and FPR 4:

```
# Assume FPR 5 holds 0xC053 4000 0000 0000.
# Assume FPR 4 holds 0x400C 0000 0000 0000.
# Assume CR = 0 and FPSCR = 0.
fcmpu 6,4,5
# CR now contains 0x0000 0040.
# FPSCR now contains 0x0000 4000.
```

**Related Information**

- Floating-Point Processor
- Floating-Point Compare Instructions

**fctid (Floating Convert to Integer Double Word) Instruction**

**Purpose**

Convert the contents of a floating-point register to a 64-bit signed fixed-point integer, placing the results into another floating-point register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>00000</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-30</td>
<td>814</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
PowerPC
fctid  \( \text{FRT FRB (Rc=0)} \)
fctid.  \( \text{FRT FRB (Rc=1)} \)

**Description**
The floating-point operand in floating-point register (FPR) \( FRB \) is converted to a 64-bit signed fixed-point integer, using the rounding mode specified by FPSCR[RN], and placed into FPR \( FRT \).

If the operand in \( FRB \) is greater than \( 2^{63} - 1 \), then FPR \( FRT \) is set to 0x7FFF_FFFF_FFFF_FFFF. If the operand in \( FRB \) is less than \( 2^{63} \), then FPR \( FRT \) is set to 0x8000_0000_0000_0000.

Except for enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

The \text{fctid} \ instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fctid</td>
<td>FPRF(undefined),FR,FI,XX,VXSNAN,VXCVI</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fctid.</td>
<td>FPRF(undefined),FR,FI,XX,VXSNAN,VXCVI</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

**Parameters**

\( FRT \)  \( \text{Specifies the target floating-point register for the operation.} \)

\( FRB \)  \( \text{Specifies the source floating-point register for the operation.} \)

**Implementation**

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

\text{**fctidz (Floating Convert to Integer Double Word with Round toward Zero) Instruction**}

**Purpose**

Convert the contents of a floating-point register to a 64-bit signed fixed-point integer using the round-toward-zero rounding mode. Place the results into another floating-point register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>00000</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-30</td>
<td>815</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
**PowerPC**

**fctidz**  
$f\text{RT}$ $f\text{RB}$ ($Rc=0$)

**fctidz.**  
$f\text{RT}$ $f\text{RB}$ ($Rc=1$)

**Description**

The floating-point operand in floating-point register (FRP) $f\text{RB}$ is converted to a 64-bit signed fixed-point integer, using the rounding mode round toward zero, and placed into FRP $f\text{RT}$.

If the operand in FRP $f\text{RB}$ is greater than $2^{63} - 1$, then FRP $f\text{RT}$ is set to $0x7FFFF_FFFF_FFFF_FFFF$.

If the operand in $f\text{RB}$ is less than $2^{63}$, then FRP $f\text{RT}$ is set to $0x8000_0000_0000_0000$.

Except for enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

The **fctidz** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fctidz</td>
<td>FPRF(undefined),FR,FI,FX,XX,VXSNAN,VXCVI</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fctidz.</td>
<td>FPRF(undefined),FR,FI,FX,XX,VXSNAN,VXCVI</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

**Parameters**

$FRT$  
Specifies the target floating-point register for the operation.

$FRB$  
Specifies the source floating-point register for the operation.

**Implementation**

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

**fctiw or fcir (Floating Convert to Integer Word) Instruction**

**Purpose**

Converts a floating-point operand to a 32-bit signed integer.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-30</td>
<td>14</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
Description
The **fctiw** and **fcir** instructions convert the floating-point operand in floating-point register (FPR) *FRB* to a 32-bit signed, fixed-point integer, using the rounding mode specified by Floating-Point Status and Control Register (FPSCR) RN. The result is placed in bits 32-63 of FPR *FRT*. Bits 0-31 of FPR *FRT* are undefined.

If the operand in FPR *FRB* is greater than 231 - 1, then the bits 32-63 of FPR *FRT* are set to 0x7FFF FFFF. If the operand in FPR *FRB* is less than -231, then the bits 32-63 of FPR *FRT* are set to 0x8000 0000.

The **fctiw** and **fcir** instruction each have two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fctiw</strong></td>
<td>C,FL,FG,FE,FU,FR,FI,FX,XX,VXCVI, VXSNAN</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><strong>fctiw.</strong></td>
<td>C,FL,FG,FE,FU,FR,FI,FX,XX,VXCVI, VXSNAN</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
<tr>
<td><strong>fcir</strong></td>
<td>C,FL,FG,FE,FU,FR,FI,FX,XX,VXCVI, VXSNAN</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><strong>fcir.</strong></td>
<td>C,FL,FG,FE,FU,FR,FI,FX,XX,VXCVI, VXSNAN</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

The syntax forms of the **fctiw** and **fcir** instructions always affect the FPSCR. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating-Point Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1. FPSCR(C,FI,FG,FE,FU) are undefined.

Parameters

- **FRT**  Specifies the floating-point register where the integer result is placed.
- **FRB**  Specifies the source floating-point register for the floating-point operand.

Examples
The following code converts a floating-point value into an integer for use as an index in an array of floating-point values:

```c
# Assume GPR 4 contains the address of the first element of
# the array.
# Assume GPR 1 contains the stack pointer.
# Assume a doubleword TEMP variable is allocated on the stack
# for use by the conversion routine.
# Assume FPR 6 contains the floating-point value for conversion
# into an index.
fctiw 5,6               # Convert floating-point value
                          # to integer.
stfd 5,TEMP(1)          # Store to temp location.
```
Related Information
- Floating-Point Processor
- Floating-Point Arithmetic Instructions
- Interpreting the Contents of a Floating-Point Register

**fctiwz or fcirz (Floating Convert to Integer Word with Round to Zero) Instruction**

**Purpose**
Converts a floating-point operand to a 32-bit signed integer, rounding the result towards 0.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-30</td>
<td>15</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**Description**
The `fctiwz` and `fcirz` instructions convert the floating-point operand in floating-point register (FPR) `FRB` to a 32-bit, signed, fixed-point integer, rounding the operand toward 0. The result is placed in bits 32-63 of FPR `FRT`. Bits 0-31 of FPR `FRT` are undefined.

If the operand in FPR `FRB` is greater than 231 - 1, then the bits 32-63 of FPR `FRT` are set to 0x7FFF FFFF. If the operand in FPR `FRB` is less than -231, then the bits 32-63 of FPR `FRT` are set to 0x8000 0000.

The `fctiwz` and `fcirz` instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 1.
The syntax forms of the `fctiwz` and `fcirz` instructions always affect the Floating-Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating-Point Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1. FPSCR(C,FI,FG,FE,FU) are undefined.

Parameters

FRT Specifies the floating-point register where the integer result is placed.
FRB Specifies the source floating-point register for the floating-point operand.

Examples

The following code adds a floating-point value to an array element selected based on a second floating-point value. If value2 is greater than or equal to n, but less than n+1, add value1 to the nth element of the array:

```
# Assume GPR 4 contains the address of the first element of # the array.
# Assume GPR 1 contains the stack pointer.
# Assume a doubleword TEMP variable is allocated on the stack # for use by the conversion routine.
# Assume FPR 6 contains value2.
# Assume FPR 4 contains value1.

fctiwz 5,6 # Convert value2 to integer.
stfd 5,TEMP(1) # Store to temp location.
lwz 3,TEMP+4(1) # Get the integer part of the # doubleword.
lfdx 5,3,4 # Get the selected array element.
fadd 5,5,4 # Add value1 to array element.
stfd 5,3,4 # Save the new value of the # array element.
```

Related Information

Floating-Point Processor.
Floating-Point Arithmetic Instructions.
Interpreting the Contents of a Floating-Point Register.

**fdiv or fd (Floating Divide) Instruction**

**Purpose**
Divides one floating-point operand by another.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
</tbody>
</table>
### Description

The `fdiv` and `fd` instructions divide the 64-bit, double-precision floating-point operand in floating-point register (FPR) `FRA` by the 64-bit, double-precision floating-point operand in FPR `FRB`. No remainder is preserved.

The `fdivs` instruction divides the 32-bit single-precision floating-point operand in FPR `FRA` by the 32-bit single-precision floating-point operand in FPR `FRB`. No remainder is preserved.

The result is rounded under control of the Floating-Point Rounding Control Field `RN` of the Floating-Point Status and Control Register (FPSCR), and is placed in the target FPR `FRT`.

The floating-point division operation is based on exponent subtraction and division of the two significands.

**Note:** If an operand is a denormalized number, then it is prenormalized before the operation is begun.

The Floating-Point Result Flags Field of the Floating-Point Status and Control Register is set to the class and sign of the result, except for Invalid Operation Exceptions, when the Floating-Point Invalid Operation Exception Enable bit is 1.
The `fdiv`, `fdivs`, and `fd` instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdiv</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, ZX,XX,VSXSNAN, VXIDI, VXZDZ</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fdiv.</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, ZX,XX,VSXSNAN, VXIDI, VXZDZ</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
<tr>
<td>fdivs</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, ZX,XX,VSXSNAN, VXIDI, VXZDZ</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fdivs.</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, ZX,XX,VSXSNAN, VXIDI, VXZDZ</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
<tr>
<td>fd</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, ZX,XX,VSXSNAN, VXIDI, VXZDZ</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fd.</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, ZX,XX,VSXSNAN, VXIDI, VXZDZ</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

All syntax forms of the `fdiv`, `fdivs`, and `fd` instructions always affect the Floating-Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating-Point Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

**Parameters**

- **FRT** Specifies target floating-point register for operation.
- **FRA** Specifies source floating-point register containing the dividend.
- **FRB** Specifies source floating-point register containing the divisor.

**Examples**

1. The following code divides the contents of FPR 4 by the contents of FPR 5, places the result in FPR 6, and sets the Floating-Point Status and Control Register to reflect the result of the operation:

   ```
   # Assume FPR 4 contains 0xC053 4000 0000 0000.
   # Assume FPR 5 contains 0x400C 0000 0000 0000.
   # Assume FPSCR = 0.
   fdiv 6,4,5
   # FPR 6 now contains 0xC036 0000 0000 0000.
   # FPSCR now contains 0x0000 8000.
   ```

2. The following code divides the contents of FPR 4 by the contents of FPR 5, places the result in FPR 6, and sets Condition Register Field 1 and the Floating-Point Status and Control Register to reflect the result of the operation:

   ```
   # Assume FPR 4 contains 0xC053 4000 0000 0000.
   # Assume FPR 5 contains 0x400C 0000 0000 0000.
   # Assume FPSCR = 0.
   fdiv 6,4,5
   # FPR 6 now contains 0xC036 0000 0000 0000.
   # FPSCR now contains 0x0000 8000.
   # CR contains 0x0000 0000.
   ```

**Related Information**

- Floating-Point Processor
- Floating-Point Arithmetic Instructions
Interpreting the Contents of a Floating-Point Register.

**fmadd or fma (Floating Multiply-Add) Instruction**

**Purpose**
Adds one floating-point operand to the result of multiplying two floating-point operands without an intermediate rounding operation.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-25</td>
<td>FRC</td>
</tr>
<tr>
<td>26-30</td>
<td>29</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**Description**
The **fmadd** and **fma** instructions multiply the 64-bit, double-precision floating-point operand in floating-point register (FPR) **FRA** by the 64-bit, double-precision floating-point operand in FPR **FRC**, and then add the result of this operation to the 64-bit, double-precision floating-point operand in FPR **FRB**.
The **fmadds** instruction multiplies the 32-bit, single-precision floating-point operand in FPR FRA by the 32-bit, single-precision floating-point operand in FPR FRC and adds the result of this operation to the 32-bit, single-precision floating-point operand in FPR FRB.

The result is rounded under control of the Floating-Point Rounding Control Field RN of the Floating-Point Status and Control Register and is placed in the target FPR FRT.

**Note:** If an operand is a denormalized number, then it is prenormalized before the operation is begun.

The Floating-Point Result Flags Field of the Floating-Point Status and Control Register is set to the class and sign of the result, except for Invalid Operation Exceptions, when the Floating-Point Invalid Operation Exception Enable bit is 1.

The **fmadd**, **fmadds**, and **fm** instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmadd</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fmadd.</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
<tr>
<td>fmadds</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fmadds.</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
<tr>
<td>fma</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fma.</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

All syntax forms of the **fmadd**, **fmadds**, and **fm** instructions always affect the Floating-Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating-Point Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

**Parameters**

- **FRT** Specifies target floating-point register for operation.
- **FRA** Specifies source floating-point register containing a multiplier.
- **FRB** Specifies source floating-point register containing the addend.
- **FRC** Specifies source floating-point register containing a multiplier.

**Examples**

1. The following code multiplies the contents of FPR 4 and FPR 5, adds the contents of FPR 7, places the result in FPR 6, and sets the Floating-Point Status and Control Register to reflect the result of the operation:

```plaintext
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume FPR 7 contains 0x3DE2 6AB4 B33C 110A.
# Assume FPSCR = 0.
fmadd 6,4,5,7
# FPR 6 now contains 0xC070 07FF FFFF F6CB.
# FPSCR now contains 0x8206 8000.
```

2. The following code multiplies the contents of FPR 4 and FPR 5, adds the contents of FPR 7, places the result in FPR 6, and sets the Floating-Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

```plaintext
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume FPR 7 contains 0x3DE2 6AB4 B33C 110A.
# Assume FPSCR = 0.
fmadd 6,4,5,7
# FPR 6 now contains 0xC070 07FF FFFF F6CB.
# FPSCR now contains 0x8206 8000.
```
Related Information

Floating-Point Processor.

Interpreting the Contents of a Floating-Point Register.

fmr (Floating Move Register) Instruction

Purpose
Copies the contents of one floating-point register into another floating-point register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-30</td>
<td>72</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

fmr FRT FRB
fmr. FRT FRB

Description
The fmr instruction places the contents of floating-point register (FPR) FRB into the target FPR FRT.

The fmr instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmr</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fmr.</td>
<td>None</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

The two syntax forms of the fmr instruction never affect the Floating-Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating-Point Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

Parameters

FRT Specifies target floating-point register for operation.
FRB  Specifies source floating-point register for operation.

Examples

1. The following code copies the contents of FPR 4 into FPR 6 and sets the Floating-Point Status and Control Register to reflect the result of the operation:

   Asssume FPR 4 contains 0xC053 4000 0000 0000.
   fmr 6,4
   FPR 6 now contains 0xC053 4000 0000 0000.
   FPSCR now contains 0x0000 0000.

2. The following code copies the contents of FPR 25 into FPR 6 and sets the Floating-Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

   Assume FPR 25 contains 0xFFFF FFFF FFFF FFFF.
   Assume FPSCR = 0 and CR = 0.
   fmr. 6,25
   FPR 6 now contains 0xFFFF FFFF FFFF FFFF.
   FPSCR now contains 0x0000 0000.
   CR now contains 0x0000 0000.

Related Information

Floating-Point Processor.

Interpreting the Contents of a Floating-Point Register.

Floating-Point Move Instructions.

fmsub or fms (Floating Multiply-Subtract) Instruction

Purpose

Subtracts one floating-point operand from the result of multiplying two floating-point operands without an intermediate rounding operation.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-25</td>
<td>FRC</td>
</tr>
<tr>
<td>26-30</td>
<td>28</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC

fmsub

fmsub.

POWER family

fms

fms.
The **fmsub** and **fmsub** instructions multiply the 64-bit, double-precision floating-point operand in floating-point register (FPR) *FRA* by the 64-bit, double-precision floating-point operand in FPR *FRC* and subtract the 64-bit, double-precision floating-point operand in FPR *FRB* from the result of the multiplication.

The **fmsubs** instruction multiplies the 32-bit, single-precision floating-point operand in FPR *FRA* by the 32-bit, single-precision floating-point operand in FPR *FRC* and subtracts the 32-bit, single-precision floating-point operand in FPR *FRB* from the result of the multiplication.

The result is rounded under control of the Floating-Point Rounding Control Field *RN* of the Floating-Point Status and Control Register and is placed in the target FPR *FRT*.

**Note:** If an operand is a denormalized number, then it is prenormalized before the operation is begun.

The Floating-Point Result Flags Field of the Floating-Point Status and Control Register is set to the class and sign of the result, except for Invalid Operation Exceptions, when the Floating-Point Invalid Operation Exception Enable bit is 1.

The **fmsub**, **fmsubs**, and **fms** instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fmsub</strong></td>
<td>C,FL,FG,FE,UF,FR,FI,OX,XX,VXSNAN,VXSI,VXIMZ</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><strong>fmsub.</strong></td>
<td>C,FL,FG,FE,UF,FR,FI,OX,XX,VXSNAN,VXSI,VXIMZ</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
<tr>
<td><strong>fmsubs</strong></td>
<td>C,FL,FG,FE,UF,FR,FI,OX,XX,VXSNAN,VXSI,VXIMZ</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><strong>fmsubs.</strong></td>
<td>C,FL,FG,FE,UF,FR,FI,OX,XX,VXSNAN,VXSI,VXIMZ</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
<tr>
<td><strong>fms</strong></td>
<td>C,FL,FG,FE,UF,FR,FI,OX,XX,VXSNAN,VXSI,VXIMZ</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><strong>fms.</strong></td>
<td>C,FL,FG,FE,UF,FR,FI,OX,XX,VXSNAN,VXSI,VXIMZ</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

All syntax forms of the **fmsub**, **fmsubs**, and **fms** instructions always affect the Floating-Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point
Exception (FX), Floating-Point Enabled Exception (FEX), Floating-Point Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

**Parameters**

- **FRT** Specifies target floating-point register for operation.
- **FRA** Specifies source floating-point register containing a multiplier.
- **FRB** Specifies source floating-point register containing the quantity to be subtracted.
- **FRC** Specifies source floating-point register containing a multiplier.

**Examples**

1. The following code multiplies the contents of FPR 4 and FPR 5, subtracts the contents of FPR 7 from the product of the multiplication, places the result in FPR 6, and sets the Floating-Point Status and Control Register to reflect the result of the operation:

   ```
   # Assume FPR 4 contains 0xC053 4000 0000 0000.
   # Assume FPR 5 contains 0x400C 0000 0000 0000.
   # Assume FPR 7 contains 0x3DE2 6AB4 B33c 110A.
   # Assume FPSCR = 0.
   fmsub 6,4,5,7
   # FPR 6 now contains 0xC070 D800 0000 0935.
   # FPSCR now contains 0x8202 8000.
   ```

2. The following code multiplies the contents of FPR 4 and FPR 5, subtracts the contents of FPR 7 from the product of the multiplication, places the result in FPR 6, and sets the Floating-Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

   ```
   # Assume FPR 4 contains 0xC053 4000 0000 0000.
   # Assume FPR 5 contains 0x400C 0000 0000 0000.
   # Assume FPR 7 contains 0x3DE2 6AB4 B33c 110A.
   # Assume FPSCR = 0 and CR = 0.
   fmsub 6,4,5,7
   # FPR 6 now contains 0xC070 D800 0000 0935.
   # FPSCR now contains 0x8202 8000.
   # CR now contains 0x0800 0000.
   ```

**Related Information**

- [Floating-Point Processor](#)
- [Interpreting the Contents of a Floating-Point Register](#)

**fmul or fm (Floating Multiply) Instruction**

**Purpose**

Multiplies two floating-point operands.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-25</td>
<td>FRC</td>
</tr>
<tr>
<td>26-30</td>
<td>25</td>
</tr>
</tbody>
</table>
Bits | Value
--- | ---
31 | Rc

PowerPC

fmul | FRT FRA FRC
fmuls | FRT FRA FRC

POWER family

fm | FRT FRA FRC
fm | FRT FRA FRC

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>59</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-25</td>
<td>FRC</td>
</tr>
<tr>
<td>26-30</td>
<td>25</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC

fmuls | FRT FRA FRC
fmuls | FRT FRA FRC

Description

The `fmul` and `fm` instructions multiply the 64-bit, double-precision floating-point operand in floating-point register (FPR) `FRA` by the 64-bit, double-precision floating-point operand in FPR `FRC`.

The `fmuls` instruction multiplies the 32-bit, single-precision floating-point operand in FPR `FRA` by the 32-bit, single-precision floating-point operand in FPR `FRC`.

The result is rounded under control of the Floating-Point Rounding Control Field `RN` of the Floating-Point Status and Control Register and is placed in the target FPR `FRT`.

Multiplication of two floating-point numbers is based on exponent addition and multiplication of the two significands.

**Note:** If an operand is a denormalized number, then it is prenormalized before the operation is begun.

The Floating-Point Result Flags Field of the Floating-Point Status and Control Register is set to the class and sign of the result, except for Invalid Operation Exceptions, when the Floating-Point Invalid Operation Exception Enable bit is 1.

The `fmul`, `fmuls`, and `fm` instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 1.
All syntax forms of the \texttt{fmul}, \texttt{fmuls}, and \texttt{fm} instructions always affect the Floating-Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating-Point Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

**Parameters**

- \textit{FRT} \hspace{1em} Specifies target floating-point register for operation.
- \textit{FRA} \hspace{1em} Specifies source floating-point register for operation.
- \textit{FRC} \hspace{1em} Specifies source floating-point register for operation.

**Examples**

1. The following code multiplies the contents of FPR 4 and FPR 5, places the result in FPR 6, and sets the Floating-Point Status and Control Register to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume FPSCR = 0.
fmul 6,4,5
# FPR 6 now contains 0xC070 D800 0000 0000.
# FPSCR now contains 0x0000 8000.
```

2. The following code multiplies the contents of FPR 4 and FPR 25, places the result in FPR 6, and sets Condition Register Field 1 and the Floating-Point Status and Control Register to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 25 contains 0xFFFF FFFF FFFF FFFF.
# Assume FPSCR = 0 and CR = 0.
fmul. 6,4,25
# FPR 6 now contains 0xFFFF FFFF FFFF FFFF.
# FPSCR now contains 0x0001 1000.
# CR now contains 0x0000 0000.
```

**Related Information**

- Floating-Point Processor.
- Floating-Point Arithmetic Instructions.
- Interpreting the Contents of a Floating-Point Register.
fnabs (Floating Negative Absolute Value) Instruction

Purpose
Negates the absolute contents of a floating-point register and places the result in another floating-point register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-30</td>
<td>136</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

fnabs \[ FRT,FRB \]
fnabs \[ FRT,FRB \]

Description
The fnabs instruction places the negative absolute of the contents of floating-point register (FPR) \( FRB \) with bit 0 set to 1 into the target FPR \( FRT \).

The fnabs instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fnabs</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fnabs.</td>
<td>None</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

The two syntax forms of the fnabs instruction never affect the Floating-Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating-Point Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

Parameters

\( FRT \)  Specifies target floating-point register for operation.
\( FRB \)  Specifies source floating-point register for operation.

Examples
1. The following code negates the absolute contents of FPR 5 and places the result into FPR 6:
   
   ```
   # Assume FPR 5 contains 0x400C 0000 0000 0000.
   fnabs 6,5
   # FPR 6 now contains 0xC00C 0000 0000 0000.
   ```

2. The following code negates the absolute contents of FPR 4, places the result into FPR 6, and sets Condition Register Field 1 to reflect the result of the operation:
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume CR = 0.
fnabs. 6,4
# FPR 6 now contains 0xC053 4000 0000 0000.
# CR now contains 0x0.

**Related Information**

- Floating-Point Processor
- Floating-Point Move Instructions
- Interpreting the Contents of a Floating-Point Register

---

**fneg (Floating Negate) Instruction**

**Purpose**

Negates the contents of a floating-point register and places the result into another floating-point register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-30</td>
<td>40</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

```
fneg
fneg.
```

**Description**

The `fneg` instruction places the negated contents of floating-point register `FRB` into the target FPR `FRT`.

The `fneg` instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fneg</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fneg.</td>
<td>None</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

The two syntax forms of the `fneg` instruction never affect the Floating-Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating-Point Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

**Parameters**

- `FRT` Specifies target floating-point register for operation.
FRB  Specifies source floating-point register for operation.

Examples
1. The following code negates the contents of FPR 5 and places the result into FPR 6:
   
   ```
   # Assume FPR 5 contains 0x400C 0000 0000 0000.
   fneg 6,5
   # FPR 6 now contains 0xC00C 0000 0000 0000.
   ```

2. The following code negates the contents of FPR 4, places the result into FPR 6, and sets Condition Register Field 1 to reflect the result of the operation:

   ```
   # Assume FPR 4 contains 0xC053 4000 0000 0000.
   fneg. 6,4
   # FPR 6 now contains 0x4053 4000 0000 0000.
   # CR now contains 0x0000 0000.
   ```

Related Information
- Floating-Point Processor
- Floating-Point Move Instructions
- Interpreting the Contents of a Floating-Point Register

fnmadd or fnma (Floating Negative Multiply-Add) Instruction

Purpose
Multiplies two floating-point operands, adds the result to one floating-point operand, and places the negative of the result in a floating-point register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-25</td>
<td>FRC</td>
</tr>
<tr>
<td>26-30</td>
<td>31</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC
- fnmadd
- fnmadd.

POWER family
- fnma
- fnma.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>59</td>
</tr>
<tr>
<td>Bits</td>
<td>Value</td>
</tr>
<tr>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-25</td>
<td>FRC</td>
</tr>
<tr>
<td>26-30</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>Rc</td>
</tr>
</tbody>
</table>

### Description

The **fnmadd** and **fnma** instructions multiply the 64-bit, double-precision floating-point operand in floating-point register (FPR) *FRA* by the 64-bit, double-precision floating-point operand in FPR *FRC*, and add the 64-bit, double-precision floating-point operand in FPR *FRB* to the result of the multiplication.

The **fnmadds** instruction multiplies the 32-bit, single-precision floating-point operand in FPR *FRA* by the 32-bit, single-precision floating-point operand in FPR *FRC*, and adds the 32-bit, single-precision floating-point operand in FPR *FRB* to the result of the multiplication.

The result of the addition is rounded under control of the Floating-Point Rounding Control Field *RN* of the Floating-Point Status and Control Register.

**Note:** If an operand is a denormalized number, then it is prenormalized before the operation is begun.

The **fnmadd** and **fnma** instructions are identical to the **fmadd** and **fma** (Floating Multiply-Add Single) instructions with the final result negated, but with the following exceptions:

- Quiet NaNs (QNaNs) propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- Signaling NaNs (SNaNs) that are converted to QNaNs as the result of a disabled Invalid Operation Exception have no effect on their “sign” bit.

The Floating-Point Result Flags Field of the Floating-Point Status and Control Register is set to the class and sign of the result, except for Invalid Operation Exceptions, when the Floating-Point Invalid Operation Exception Enable bit is 1.

The **fnmadd**, **fnmadds**, and **fnma** instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 1.
All syntax forms of the \texttt{fnmad}, \texttt{fnmadds}, and \texttt{fnma} instructions always affect the Floating-Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating-Point Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

\textbf{Note:} Rounding occurs before the result of the addition is negated. Depending on $RN$, an inexact value may result.

\section*{Parameters}

\begin{itemize}
  \item \texttt{FRT} Specifies target floating-point register for operation.
  \item \texttt{FRA} Specifies source floating-point register for operation.
  \item \texttt{FRB} Specifies source floating-point register for operation.
  \item \texttt{FRC} Specifies source floating-point register for operation.
\end{itemize}

\section*{Examples}

1. The following code multiplies the contents of FPR 4 and FPR 5, adds the result to the contents of FPR 7, stores the negated result in FPR 6, and sets the Floating-Point Status and Control Register to reflect the result of the operation:

\begin{verbatim}
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume FPR 7 contains 0x3DE2 6AB4 B33c 110A.
# Assume FPSCR = 0.
fnmadd 6,4,5,7
# FPR 6 now contains 0x4070 D7FF FFFF F6CB.
# FPSCR now contains 0x8206 4000.
\end{verbatim}

2. The following code multiplies the contents of FPR 4 and FPR 5, adds the result to the contents of FPR 7, stores the negated result in FPR 6, and sets the Floating-Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

\begin{verbatim}
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume FPR 7 contains 0x3DE2 6AB4 B33c 110A.
# Assume FPSCR = 0 and CR = 0.
fnmadd. 6,4,5,7
# FPR 6 now contains 0x4070 D7FF FFFF F6CB.
# FPSCR now contains 0x8206 4000.
# CR now contains 0x0800 0000.
\end{verbatim}

\section*{Related Information}

- Floating-Point Processor
- Interpreting the Contents of a Floating-Point Register

\section*{fnmsub or fnms (Floating Negative Multiply-Subtract) Instruction}

\section*{Purpose}

Multiplies two floating-point operands, subtracts one floating-point operand from the result, and places the negative of the result in a floating-point register.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-25</td>
<td>FRC</td>
</tr>
<tr>
<td>26-30</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC

```
fnmsub    FRT FRA FRC FRB
fnmsub.   FRT FRA FRC FRB
```

POWER family

```
fnms      FRT FRA FRC FRB
fnms.     FRT FRA FRC FRB
```

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>59</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-25</td>
<td>FRC</td>
</tr>
<tr>
<td>26-30</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC

```
fnmsubs  FRT FRA FRC FRB
fnmsubs. FRT FRA FRC FRB
```

Description

The `fnms` and `fnmsub` instructions multiply the 64-bit, double-precision floating-point operand in floating-point register (FPR) `FRA` by the 64-bit double-precision floating-point operand in FPR `FRC`, subtract the 64-bit, double-precision floating-point operand in FPR `FRB` from the result of the multiplication, and place the negated result in the target FPR `FRT`.

The `fnmsubs` instruction multiplies the 32-bit, single-precision floating-point operand in FPR `FRA` by the 32-bit, single-precision floating-point operand in FPR `FRC`, subtracts the 32-bit, single-precision floating-point operand in FPR `FRB` from the result of the multiplication, and places the negated result in the target FPR `FRT`.

The subtraction result is rounded under control of the Floating-Point Rounding Control Field `RN` of the Floating-Point Status and Control Register.
**Note:** If an operand is a denormalized number, then it is prenormalized before the operation is begun.

The **fnms** and **fnmsub** instructions are identical to the **fmsub** and **fms** (Floating Multiply-Subtract Single) instructions with the final result negated, but with the following exceptions:

- Quiet NaNs (QNaNs) propagate with no effect on their "sign" bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a "sign" bit of zero.
- Signaling NaNs (SNaNs) that are converted to QNaNs as the result of a disabled Invalid Operation Exception have no effect on their "sign" bit.

The Floating-Point Result Flags Field of the Floating-Point Status and Control Register is set to the class and sign of the result, except for Invalid Operation Exceptions, when the Floating-Point Invalid Operation Exception Enable bit is 1.

The **fnmsub**, **fnmsubs**, and **fnms** instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fnmsub</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX,XX,VXSNAN,VXISI,VXIMZ</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fnmsubs</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX,XX,VXSNAN,VXISI,VXIMZ</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
<tr>
<td>fnms</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX,XX,VXSNAN,VXISI,VXIMZ</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fnmsubs.</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX,XX,VXSNAN,VXISI,VXIMZ</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
<tr>
<td>fnms.</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX,XX,VXSNAN,VXISI,VXIMZ</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fnmsubs.</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX,XX,VXSNAN,VXISI,VXIMZ</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

All syntax forms of the **fnmsub**, **fnmsubs**, and **fnms** instructions always affect the Floating-Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating-Point Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

**Note:** Rounding occurs before the result of the addition is negated. Depending on $RN$, an inexact value may result.

**Parameters**

- $FRT$ Specifies target floating-point register for operation.
- $FRA$ Specifies first source floating-point register for operation.
- $FRB$ Specifies second source floating-point register for operation.
- $FRC$ Specifies third source floating-point register for operation.

**Examples**

1. The following code multiplies the contents of FPR 4 and FPR 5, subtracts the contents of FPR 7 from the result, stores the negated result in FPR 6, and sets the Floating-Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

```assembly
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume FPR 7 contains 0x3DE2 6AB4 B33c 110A.
```
2. The following code multiplies the contents of FPR 4 and FPR 5, subtracts the contents of FPR 7 from the result, stores the negated result in FPR 6, and sets the Floating-Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume FPR 7 contains 0x3DE2 6A84 B33c 110A.
# Assume FPSCR = 0 and CR = 0.
fnmsub. 6,4,5,7
# FPR 6 now contains 0x4070 D800 0000 0935.
# FPSCR now contains 0x8202 4000.
# CR now contains 0x0800 0000.
```

Related Information

- Floating-Point Processor
- Interpreting the Contents of a Floating-Point Register

**fres (Floating Reciprocal Estimate Single) Instruction**

**Purpose**

Calculates a single-precision estimate of the reciprocal of a floating-point operand.

**Note:** The fres instruction is defined only in the PowerPC architecture and is an optional instruction. It is supported on the PowerPC 603 RISC Microprocessor, and PowerPC 604 RISC Microprocessor, but not supported on the PowerPC 601 RISC Microprocessor.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>59</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-25</td>
<td>///</td>
</tr>
<tr>
<td>26-30</td>
<td>24</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**Description**

The fres instruction calculates a single-precision estimate of the reciprocal of the 64-bit, double-precision floating-point operand in floating-point register (FPR) FRB and places the result in FPR FRT.
The estimate placed into register \textit{FRT} is correct to a precision of one part in 256 of the reciprocal of \textit{FRB}. The value placed into \textit{FRT} may vary between implementations, and between different executions on the same implementation.

The following table summarizes special conditions:

<table>
<thead>
<tr>
<th>Special Conditions</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative Infinity</td>
<td>Negative 0</td>
<td>None</td>
</tr>
<tr>
<td>Negative 0</td>
<td>Negative Infinity(^1)</td>
<td>ZX</td>
</tr>
<tr>
<td>Positive 0</td>
<td>Positive Infinity(^1)</td>
<td>ZX</td>
</tr>
<tr>
<td>Positive Infinity</td>
<td>Positive 0</td>
<td>None</td>
</tr>
<tr>
<td>\text{SNaN}</td>
<td>\text{QNaN}(^2)</td>
<td>\text{VXSNAN}</td>
</tr>
<tr>
<td>\text{QNaN}</td>
<td>\text{QNaN}</td>
<td>None</td>
</tr>
</tbody>
</table>

\(^1\)No result if FPSCRZE = 1.
\(^2\)No result if FPSCRVE = 1.

FPSCRFPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCRVE = 1 and Zero Divide Exceptions when FPSCRZE = 1.

The \textit{fres} instruction has two syntax forms. Both syntax forms always affect the FPSCR register. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{fres}</td>
<td>\text{C,FL,FG,FE,FU,FR,FI,FX,OX, UX,ZX,VXSNAN}</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>\textit{fres.}</td>
<td>\text{C,FL,FG,FE,FU,FR,FI,FX,OX, UX,ZX,VXSNAN}</td>
<td>1</td>
<td>\text{FX,FEX,VX,OX}</td>
</tr>
</tbody>
</table>

The \textit{fres.} syntax form sets the Record (Rc) bit to 1; and the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating-Point Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1 (CR1). The \textit{fres} syntax form sets the Record (Rc) bit to 0 and does not affect Condition Register Field 1 (CR1).

\textbf{Parameters}

\textit{FRT} \hspace{1em} Specifies target floating-point register for operation.
\textit{FRB} \hspace{1em} Specifies source floating-point register for operation.

\textbf{Related Information}

- \textbf{Floating-Point Processor}
- \textbf{Floating-Point Arithmetic Instructions}
- \textbf{Interpreting the Contents of a Floating-Point Register}
frsp (Floating Round to Single Precision) Instruction

Purpose
Rounds a 64-bit, double precision floating-point operand to single precision and places the result in a floating-point register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-30</td>
<td>12</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

frsp
frsp.

Description
The frsp instruction rounds the 64-bit, double-precision floating-point operand in floating-point register (FPR) FRB to single precision, using the rounding mode specified by the Floating Rounding Control field of the Floating-Point Status and Control Register, and places the result in the target FPR FRT.

The Floating-Point Result Flags Field of the Floating-Point Status and Control Register is set to the class and sign of the result, except for Invalid Operation (SNaN), when Floating-Point Status and Control Register Floating-Point Invalid Operation Exception Enable bit is 1.

The frsp instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>frsp</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX,XX,VXSNAN</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>frsp.</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX,XX,VXSNAN</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

The two syntax forms of the frsp instruction always affect the Floating-Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating-Point Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

Notes:
1. The frsp instruction uses the target register of a previous floating-point arithmetic operation as its source register (FRB). The frsp instruction is said to be dependent on the preceding floating-point arithmetic operation when it uses this register for source.
2. Less than two nondependent floating-point arithmetic operations occur between the frsp instruction and the operation on which it is dependent.
3. The magnitude of the double-precision result of the arithmetic operation is less than 2**128 before rounding.
4. The magnitude of the double-precision result after rounding is exactly $2^{128}$.

**Error Result**

If the error occurs, the magnitude of the result placed in the target register $FRT$ is $2^{128}$:

* $X'47F0000000000000'$ or $X'C7F0000000000000'$

This is not a valid single-precision value. The settings of the Floating-Point Status and Control Register and the Condition Register will be the same as if the result does not overflow.

**Avoiding Errors**

If the above error will cause significant problems in an application, either of the following two methods can be used to avoid the error.

1. Place two nondependent floating-point operations between a floating-point arithmetic operation and the dependent frsp instruction. The target registers for these nondependent floating-point operations should not be the same register that the frsp instruction uses as source register $FRB$.

2. Insert two frsp operations when the frsp instruction may be dependent on an arithmetic operation that precedes it by less than three floating-point instructions.

Either solution will degrade performance by an amount dependent on the particular application.

**Parameters**

$FRT$  
Specifies target floating-point register for operation.

$FRB$  
Specifies source floating-point register for operation.

**Examples**

1. The following code rounds the contents of FPR 4 to single precision, places the result in a FPR 6, and sets the Floating-Point Status and Control Register to reflect the result of the operation:

   ```
   # Assume FPR 4 contains 0xC053 4000 0000 0000.
   # Assume FPSCR = 0.
   frsp 6,4
   # FPR 6 now contains 0xC053 4000 0000 0000.
   # FPSCR now contains 0x0000 8000.
   ```

2. The following code rounds the contents of FPR 4 to single precision, places the result in a FPR 6, and sets the Floating-Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

   ```
   # Assume CR contains 0x0000 0000.
   # Assume FPR 4 contains 0xFFFF FFFF FFFF FFFF.
   # Assume FPSCR = 0.
   frsp 6,4
   # FPR 6 now contains 0xFFFF FFFF E000 0000.
   # FPSCR now contains 0x0001 1000.
   # CR now contains 0x0000 0000.
   ```

**Related Information**

[Floating-Point Processor](#)

[Interpreting the Contents of a Floating-Point Register](#)

[Floating-Point Arithmetic Instructions](#)
frsqrte (Floating Reciprocal Square Root Estimate) Instruction

Purpose
Calculates a double-precision estimated value of the reciprocal of the square root of a floating-point operand.

Note: The frsqrte instruction is defined only in the PowerPC architecture and is an optional instruction. It is supported on the PowerPC 603 RISC Microprocessor and the PowerPC 604 RISC Microprocessor, but not supported on the PowerPC 601 RISC Microprocessor.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-25</td>
<td>///</td>
</tr>
<tr>
<td>26-30</td>
<td>26</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC
frsqrte
frsqrte.

Description
The frsqrte instruction computes a double-precision estimate of the reciprocal of the square root of the 64-bit, double-precision floating-point operand in floating-point register (FPR) FRB and places the result in FPR FRT.

The estimate placed into register FRT is correct to a precision of one part in 32 of the reciprocal of the square root of FRB. The value placed in FRT may vary between implementations and between different executions on the same implementation.

The following table summarizes special conditions:

<table>
<thead>
<tr>
<th>Special Conditions</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negative Infinity</td>
<td>QNaN(^1)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>Less Than 0</td>
<td>QNaN(^1)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>Negative 0</td>
<td>Negative Infinity(^2)</td>
<td>ZX</td>
</tr>
<tr>
<td>Positive 0</td>
<td>Positive Infinity(^2)</td>
<td>ZX</td>
</tr>
<tr>
<td>Positive Infinity</td>
<td>Positive 0</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN(^1)</td>
<td>VXSQNaN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

\(^1\)No result if FPSCRVE = 1.
2No result if FPSCRZE = 1.

FPSCRFPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCRVE = 1 and Zero Divide Exceptions when FPSCRZE = 1.

The `frsqrte` instruction has two syntax forms. Both syntax forms always affect the FPSCR. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>frsqrte</code></td>
<td>C,FL,FG,FE,FR,FI,FX,ZX, VXSNAN,VXSQRT</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>frsqrte</code></td>
<td>C,FL,FG,FE,FR,FI,FX,ZX, VXSNAN,VXSQRT</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

The `frstrte` syntax form sets the Record (Rc) bit to 1; and the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating-Point Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1 (CR1). The `frstrte` syntax form sets the Record (Rc) bit to 0; and the instruction does not affect Condition Register Field 1 (CR1).

**Parameters**

- **FRT** Specified target floating-point register for operation.
- **FRB** Specifies source floating-point register for operation.

**Related Information**

- Floating-Point Processor
- Floating-Point Arithmetic Instructions
- Interpreting the Contents of a Floating-Point Register

**fsel (Floating-Point Select) Instruction**

**Purpose**

Puts either of two floating-point operands into the target register based on the results of comparing another floating-point operand with zero.

**Note:** The `fsele` instruction is defined only in the PowerPC architecture and is an optional instruction. It is supported on the PowerPC 603 RISC Microprocessor and the PowerPC 604 RISC Microprocessor, but not supported on the PowerPC 601 RISC Microprocessor.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-25</td>
<td>FRC</td>
</tr>
<tr>
<td>26-30</td>
<td>23</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
Description
The double-precision floating-point operand in floating-point register (FPR) FRA is compared with the value zero. If the value in FRA is greater than or equal to zero, floating point register FRT is set to the contents of floating-point register FRC. If the value in FRA is less than zero or is a NaN, floating point register FRT is set to the contents of floating-point register FRB. The comparison ignores the sign of zero; both +0 and -0 are equal to zero.

The fesl instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>FPSCR bits</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fesl</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fesl.</td>
<td>None</td>
<td>1</td>
<td>FX, FEX, VX, OX</td>
</tr>
</tbody>
</table>

The two syntax forms of the fesl instruction never affect the Floating-Point Status and Control Register fields. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

Parameters
FRT Specifies target floating-point register for operation.
FRA Specifies floating-point register with value to be compared with zero.
FRB Specifies source floating-point register containing the value to be used if FRA is less than zero or is a NaN.
FRC Specifies source floating-point register containing the value to be used if FRA is greater than or equal to zero.

Related Information
Floating-Point Processor.
Interpreting the Contents of a Floating-Point Register.

fsqrt (Floating Square Root, Double-Precision) Instruction
Purpose
Calculate the square root of the contents of a floating-point register, placing the result in a floating-point register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>00000</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
</tbody>
</table>
**Description**

The square root of the operand in floating-point register (FPR) $FRB$ is placed into register FPR $FRT$.

If the most-significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register FPR $FRT$.

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>- infinity</td>
<td>QNaN*</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>QNaN*</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>- 0</td>
<td>- 0</td>
<td>None</td>
</tr>
<tr>
<td>+ infinity</td>
<td>+ infinity</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN*</td>
<td>VXSQAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

Notes: * No result if FPSCR[VE] = 1

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

The $fsqrt$ instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$fsqrt$</td>
<td>FPRF,FR,FI,FX,XX,VXSNAN,VXSQRT</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>$fsqrt.$</td>
<td>FPRF,FR,FI,FX,XX,VXSNAN,VXSQRT</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

**Parameters**

- $FRT$  \(\text{specifies the target floating-point register for the operation.}\)
- $FRB$  \(\text{specifies the source floating-point register for the operation.}\)

**Implementation**

This instruction is optionally defined for PowerPC implementations. Using it on an implementation that does not support this instruction will cause the system illegal instruction error handler to be invoked.

This instruction is an optional instruction of the PowerPC architecture and may not be implemented in all machines.
fsqrts (Floating Square Root Single) Instruction

Purpose
Calculate the single-precision square root of the contents of a floating-point register, placing the result in a floating-point register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>59</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>00000</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-25</td>
<td>00000</td>
</tr>
<tr>
<td>26-30</td>
<td>22</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC
fsqrts. (Rc=0)
fsqrts. (Rc=1)

Description
The square root of the floating-point operand in floating-point register (FPR) FRB is placed into register FPR FRT.

If the most-significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register FPR FRT.

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>- infinity</td>
<td>QNaN*</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>QNaN*</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>- 0</td>
<td>- 0</td>
<td>None</td>
</tr>
<tr>
<td>+ infinity</td>
<td>+ infinity</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN*</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

Notes: * No result if FPSCR[VE] = 1

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

The fsqrts instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsqrts</td>
<td>FPRF,FR,FI,FX,XX,VXSNAN,VXSQRT</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>
Parameters

FRT  Specifies the target floating-point register for the operation.
FRB  Specifies the source floating-point register for the operation.

Implementation

This instruction is optionally defined for PowerPC implementations. Using it on an implementation that
does not support this instruction will cause the system illegal instruction error handler to be invoked.

This instruction is an optional instruction of the PowerPC architecture and may not be implemented in all
machines.

fsub or fs (Floating Subtract) Instruction

Purpose

Subtracts one floating-point operand from another and places the result in a floating-point register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-25</td>
<td>///</td>
</tr>
<tr>
<td>26-30</td>
<td>20</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC

fsub

```
FRT FRA FRB
```

fs

```
FRT FRA FRB
```

Bits  | Value |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>59</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>FRA</td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-25</td>
<td>///</td>
</tr>
<tr>
<td>26-30</td>
<td>20</td>
</tr>
</tbody>
</table>
PowerPC

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**Description**

The `fsub` and `fs` instructions subtract the 64-bit, double-precision floating-point operand in floating-point register (FPR) `FRB` from the 64-bit, double-precision floating-point operand in FPR `FRA`.

The `fsubs` instruction subtracts the 32-bit single-precision floating-point operand in FPR `FRB` from the 32-bit single-precision floating-point operand in FPR `FRA`.

The result is rounded under control of the Floating-Point Rounding Control Field `RN` of the Floating-Point Status and Control Register and is placed in the target FPR `FRT`.

The execution of the `fsub` instruction is identical to that of `fadd`, except that the contents of FPR `FRB` participate in the operation with bit 0 inverted.

The execution of the `fs` instruction is identical to that of `fa`, except that the contents of FPR `FRB` participate in the operation with bit 0 inverted.

The Floating-Point Result Flags Field of the Floating-Point Status and Control Register is set to the class and sign of the result, except for Invalid Operation Exceptions, when the Floating-Point Invalid Operation Exception Enable bit is 1.

The `fsub`, `fsubs`, and `fs` instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Floating-Point Status and Control Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsub</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXSI</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fsubs</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXSI</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
<tr>
<td>fs</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXSI</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>fs.</td>
<td>C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXSI</td>
<td>1</td>
<td>FX,FEX,VX,OX</td>
</tr>
</tbody>
</table>

All syntax forms of the `fsub`, `fsubs`, and `fs` instructions always affect the Floating-Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating-Point Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.
Parameters

FRT  Specifies target floating-point register for operation.
FRA  Specifies source floating-point register for operation.
FRB  Specifies source floating-point register for operation.

Examples

1. The following code subtracts the contents of FPR 5 from the contents of FPR 4, places the result in FPR 6, and sets the Floating-Point Status and Control Register to reflect the result of the operation:

   # Assume FPR 4 contains 0xC053 4000 0000 0000.
   # Assume FPR 5 contains 0x400C 0000 0000 0000.
   # Assume FPSCR = 0.
   fsub 6,4,5
   # FPR 6 now contains 0xC054 2000 0000 0000.
   # FPSCR now contains 0x0000 8000.

2. The following code subtracts the contents of FPR 5 from the contents of FPR 4, places the result in FPR 6, and sets the Floating-Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

   # Assume FPR 4 contains 0xC053 4000 0000 0000.
   # Assume FPR 5 contains 0x400C 0000 0000 0000.
   # Assume FPSCR = 0 and CR = 0.
   fsub. 6,5,4
   # FPR 6 now contains 0x4054 2000 0000 0000.
   # FPSCR now contains 0x0000 4000.
   # CR now contains 0x0000 0000.

Related Information

Floating-Point Processor.
Floating-Point Arithmetic Instructions.
Interpreting the Contents of a Floating-Point Register.

icbi (Instruction Cache Block Invalidate) Instruction

Purpose

Invalidates a block containing the byte addressed in the instruction cache, causing subsequent references to retrieve the block from main memory.

Note: The icbi instruction is supported only in the PowerPC architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>//</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>982</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>
Description

The icbi instruction invalidates a block containing the byte addressed in the instruction cache. If RA is not 0, the icbi instruction calculates an effective address (EA) by adding the contents of general-purpose register (GPR) RA to the contents of GPR RB.

Consider the following when using the icbi instruction:

- If the Data Relocate (DR) bit of the Machine State Register (MSR) is 0, the effective address is treated as a real address.
- If the MSR DR bit is 1, the effective address is treated as a virtual address. The MSR Relocate (IR) bit is ignored in this case.
- If a block containing the byte addressed by the EA is in the instruction cache, the block is made unusable so the next reference to the block is taken from main memory.

The icbi instruction has one syntax form and does not affect Condition Register Field 0 or the Fixed-Point Exception Register.

Parameters

RA  Specifies source general-purpose register for the EA calculation.
RB  Specifies source general-purpose register for the EA calculation.

Examples

The following code ensures that modified instructions are available for execution:

```assembly
# Assume GPR 3 contains a modified instruction.
# Assume GPR 4 contains the address of the memory location
# where the modified instruction will be stored.
swt 3,0(4)    # Store the modified instruction.
dcbf 0,4      # Copy the modified instruction to
              # main memory.
sync          # Ensure update is in main memory.
icbi 0,4      # Invalidate block with old instruction.
isync          # Discard prefetched instructions.
b newcde       # Go execute the new code.
```

Related Information

The clcs (Cache Line Compute Size) instruction, clf (Cache Line Flush) instruction, cli (Cache Line Invalidate) instruction, dcbf (Data Cache Block Flush) instruction, dcbl (Data Cache Block Invalidate) instruction, dcbst (Data Cache Block Store) instruction, dcbt (Data Cache Block Touch) instruction, dcbtst (Data Cache Block Touch for Store) instruction, dcbz or dclz (Data Cache Block Set to Zero) instruction, dclst (Data Cache Line Store) instruction, sync (Synchronize) or dcs (Data Cache Synchronize) instruction.

isync or ics (Instruction Synchronize) Instruction

Purpose

Refetches any instructions that might have been fetched prior to this instruction.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>19</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>150</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

PowerPC

**isync**

POWER family

**ics**

**Description**
The **isync** and **ics** instructions cause the processor to refetch any instructions that might have been fetched prior to the **isync** or **ics** instruction.

The PowerPC instruction **isync** causes the processor to wait for all previous instructions to complete. Then any instructions already fetched are discarded and instruction processing continues in the environment established by the previous instructions.

The POWER family instruction **ics** causes the processor to wait for any previous **dcs** instructions to complete. Then any instructions already fetched are discarded and instruction processing continues under the conditions established by the content of the Machine State Register.

The **isync** and **ics** instructions have one syntax form and do not affect Condition Register Field 0 or the Fixed-Point Exception Register.

**Examples**
The following code refetches instructions before continuing:

```
# Assume GPR 5 holds name.
# Assume GPR 3 holds 0x0.
name: dcbf 3,5
isync
```

**Related Information**
The **clcs** (Cache Line Compute Size) instruction, **clfi** (Cache Line Flush) instruction, **clii** (Cache Line Invalidate) instruction, **dcbf** (Data Cache Block Flush) instruction, **dcbi** (Data Cache Block Invalidate) instruction, **dcbst** (Data Cache Block Store) instruction, **dcbt** (Data Cache Block Touch) instruction, **dcbtst** (Data Cache Block Touch for Store) instruction, **dcbz** or **dclz** (Data Cache Line Set to Zero) instruction, **lcbi** (Instruction Cache Block Invalidate) instruction, **sync** (Synchronize) or **dcs** (Data Cache Synchronize) instruction.

**Processing and Storage**

**Functional Differences for POWER family and PowerPC Instructions.**
**Ibz (Load Byte and Zero) Instruction**

**Purpose**
Loads a byte of data from a specified location in memory into a general-purpose register and sets the remaining 24 bits to 0.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>34</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

```
lbz RT, D(RA)
```

**Description**
The *Ibz* instruction loads a byte in storage addressed by the effective address (EA) into bits 24-31 of the target general-purpose register (GPR) *RT* and sets bits 0-23 of GPR *RT* to 0.

If *RA* is not 0, the EA is the sum of the contents of GPR *RA* and *D*, a 16-bit, signed two’s complement integer sign-extended to 32 bits. If *RA* is 0, then the EA is *D*.

The *Ibz* instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Parameters**

- *RT* Specifies target general-purpose register where result of operation is stored.
- *D* 16-bit, signed two’s complement integer sign-extended to 32 bits for EA calculation.
- *RA* Specifies source general-purpose register for EA calculation.

**Examples**
The following code loads a byte of data from a specified location in memory into GPR 6 and sets the remaining 24 bits to 0:

```
.csect data[rw]
storage: .byte 'a
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
lbz 6,storage(5)
# GPR 6 now contains 0x0000 0061.
```

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Load and Store Instructions](#)
Ibzu (Load Byte and Zero with Update) Instruction

Purpose
Loads a byte of data from a specified location in memory into a general-purpose register, sets the remaining 24 bits to 0, and possibly places the address in a second general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>35</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

Ibzu \( RT \ D \ RA \)

Description
The Ibzu instruction loads a byte in storage addressed by the effective address (EA) into bits 24-31 of the target general-purpose register (GPR) \( RT \) and sets bits 0-23 of GPR \( RT \) to 0.

If \( RA \) is not 0, the EA is the sum of the contents of GPR \( RA \) and \( D \), a 16-bit signed two’s complement integer sign extended to 32 bits. If \( RA \) is 0, then the EA is \( D \).

If \( RA \) does not equal \( RT \) and \( RA \) does not equal 0, and the storage access does not cause an Alignment interrupt or a Data Storage interrupt, then the EA is stored in GPR \( RA \).

The Ibzu instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters
- \( RT \) Specifies target general-purpose register where result of operation is stored.
- \( D \) 16-bit, signed two’s complement integer sign-extended to 32 bits for EA calculation.
- \( RA \) Specifies source general-purpose register for EA calculation and possible address update.

Examples
The following code loads a byte of data from a specified location in memory into GPR 6, sets the remaining 24 bits to 0, and places the address in GPR 5:

```assembly
.csect data[rw]
storage: .byte 0x61
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
Ibzu 6,storage(5)
# GPR 6 now contains 0x0000 0061.
# GPR 5 now contains the storage address.
```

Related Information
- Fixed-Point Processor
- Fixed-Point Load and Store with Update Instructions
Ibzux (Load Byte and Zero with Update Indexed) Instruction

Purpose
Loads a byte of data from a specified location in memory into a general-purpose register, setting the remaining 24 bits to 0, and places the address in the a second general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>119</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

Ibzux   \[RT\ RA\ RB\]

Description
The Ibzux instruction loads a byte in storage addressed by the effective address (EA) into bits 24-31 of the target general-purpose register (GPR) \(RT\) and sets bits 0-23 of GPR \(RT\) to 0.

If RA is not 0, the EA is the sum of the contents of GPR RA and GPR RB. If RA is 0, then the EA is the contents of RB.

If RA does not equal RT and RA does not equal 0, and the storage access does not cause an Alignment interrupt or a Data Storage interrupt, then the EA is stored in GPR RA.

The Ibzux instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters

- **RT**: Specifies target general-purpose register where result of operation is stored.
- **RA**: Specifies source general-purpose register for EA calculation and possible address update.
- **RB**: Specifies source general-purpose register for EA calculation.

Examples
The following code loads the value located at storage into GPR 6 and loads the address of storage into GPR 5:

```plaintext
storage: .byte 0x40

# Assume GPR 5 contains 0x0000 0000.
# Assume GPR 4 is the storage address.
Ibzux 6,5,4
# GPR 6 now contains 0x0000 0040.
# GPR 5 now contains the storage address.
```
**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Load and Store with Update Instructions](#)

---

**Ibxz (Load Byte and Zero Indexed) Instruction**

**Purpose**

Loads a byte of data from a specified location in memory into a general-purpose register and sets the remaining 24 bits to 0.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>87</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

**Description**

The `Ibxz` instruction loads a byte in storage addressed by the effective address (EA) into bits 24-31 of the target general-purpose register (GPR) `RT` and sets bits 0-23 of GPR `RT` to 0.

If `RA` is not 0, the EA is the sum of the contents of GPR `RA` and GPR `RB`. If `RA` is 0, then the EA is `D`.

The `Ibxz` instruction has one syntax form and does not affect the Fixed-Point Exception Register.

**Parameters**

- `RT` Specifies target general-purpose register where result of operation is stored.
- `RA` Specifies source general-purpose register for EA calculation.
- `RB` Specifies source general-purpose register for EA calculation.

**Examples**

The following code loads the value located at storage into GPR 6:

```assembly
storage: .byte 0x61
.
# Assume GPR 5 contains 0x0000 0000.
# Assume GPR 4 is the storage address.
Ibxz 6,5,4
# GPR 6 now contains 0x0000 0061.
```

248 Assembler Language Reference
Related Information
- Fixed-Point Processor
- Fixed-Point Load and Store Instructions

**ld (Load Double Word) Instruction**

**Purpose**
Load a double-word of data into the specified general purpose register.

*This instruction should only be used on 64-bit PowerPC processors running a 64-bit application.*

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>58</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-29</td>
<td>ds</td>
</tr>
<tr>
<td>30-31</td>
<td>00</td>
</tr>
</tbody>
</table>

**PowerPC64**

```
ld  RT, D(RA)
```

**Description**
The `ld` instruction loads a double-word in storage from a specified location in memory addressed by the effective address (EA) into the target general-purpose register (GPR) `RT`.

If GPR `RA` is not 0, the EA is the sum of the contents of GPR `RA` and `D`, a 16-bit, signed two’s complement integer, fullword-aligned, sign-extended to 64 bits. If GPR `RA` is 0, then the EA is `D`.

**Parameters**
- `RT`  Specifies target general-purpose register where result of operation is stored.
- `D`   Specifies a 16-bit, signed two’s complement integer sign-extended to 32 bits for EA calculation.
- `RA`  Specifies source general-purpose register for EA calculation.

**Examples**
The following code loads a double-word from memory into GPR 4:

```
.extern mydata[RW]
.csect foodata[rw]
.local foodata[rw]
storage: .llong mydata  # address of mydata

.csect text[PR]
    # Assume GPR 5 contains address of csect foodata[RW].
ld  4,storage(5)  # GPR 5 now contains the address of mydata.
```
Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Related Information
[Fixed-Point Processor]
[Fixed-Point Load and Store Instructions]

Idarx (Store Double Word Reserve Indexed) Instruction

Purpose
This instruction should only be used on 64-bit PowerPC processors running a 64-bit application.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-30</td>
<td>84</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

PowerPC64 Idarx

Description
This instruction creates a reservation for use by a Store Double Word Conditional Indexed (stdcx.) instruction. An address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation. EA must be a multiple of eight. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Parameters

- \( rD \) Specifies source general-purpose register of stored data.
- \( rA \) Specifies source general-purpose register for EA calculation.
- \( rB \) Specifies source general-purpose register for EA calculation.
Examples

Related Information

**ldu (Store Double Word with Update) Instruction**

**Purpose**
Load a double-word of data into the specified general purpose register, updating the address base.

**This instruction should only be used on 64-bit PowerPC processors running a 64-bit application.**

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>58</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-29</td>
<td>ds</td>
</tr>
<tr>
<td>30-31</td>
<td>01</td>
</tr>
</tbody>
</table>

**PowerPC64**

**ldu**

**Description**

The **ldu** instruction loads a double-word in storage from a specified location in memory addressed by the effective address (EA) into the target general-purpose register (GPR) **RT**.

If GPR **RA** is not 0, the EA is the sum of the contents of GPR **RA** and **D**, a 16-bit, signed two's complement integer, fullword-aligned, sign-extended to 64 bits.

If **RA** = 0 or **RA** = **RT**, the instruction form is invalid.

**Parameters**

**RT** Specifies target general-purpose register where result of operation is stored.

**D** Specifies a 16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.

**RA** Specifies source general-purpose register for EA calculation.

**Examples**

The following code loads the first of 4 double-words from memory into GPR 4, incrementing GPR 5 to point to the next double-word in memory:

```
.csect    foodata[rw]
storage: .llong 5,6,7,12  # Successive double-words.
.csect    text[PR]
        # Assume GPR 5 contains address of csect foodata[RW].
ldu 4,storage(5)  # GPR 4 now contains the first double-word of
        # foodata; GPR 5 points to the second double-word.
```
Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Related Information
Fixed-Point Processor.
Fixed-Point Load and Store with Update Instructions

ldux (Store Double Word with Update Indexed) Instruction

Purpose
Load a double-word of data from a specified memory location into a general purpose register. Update the address base.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-30</td>
<td>53</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

PowerPC
ldux  

Description
The effective address (EA) is calculated from the sum of general purpose register (GPR) RA and RB. A double-word of data is read from the memory location referenced by the EA and placed into GPR RT; GRP RA is updated with the EA.

If rA = 0 or rA = rD, the instruction form is invalid.

Parameters

RT Specifies source general-purpose register of stored data.
RA Specifies source general-purpose register for EA calculation.
RB Specifies source general-purpose register for EA calculation.

Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.
Idx (Store Double Word Indexed) Instruction

Purpose
Load a double-word from a specified memory location into a general purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-30</td>
<td>21</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

PowerPC

Idx  \[ RT \ RA \ RB \]

Description
The \texttt{ldx} instruction loads a double-word from the specified memory location referenced by the effective address (EA) into the general-purpose register (GPR) \textit{RT}.

If GRP \textit{RA} is not 0, the EA is the sum of the contents of GRP \textit{RA} and \textit{B}; otherwise, the EA is equal to the contents of \textit{RB}.

Parameters
\textit{RT} Specifies target general-purpose register where result of operation is stored.
\textit{RA} Specifies source general-purpose register for EA calculation.
\textit{RB} Specifies source general-purpose register for EA calculation.

Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Ifd (Load Floating-Point Double) Instruction

Purpose
Loads a doubleword of data from a specified location in memory into a floating-point register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>50</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
</tbody>
</table>
The `lfd` instruction loads a doubleword in storage from a specified location in memory addressed by the effective address (EA) into the target floating-point register (FPR) `FRT`.

If general-purpose register (GPR) `RA` is not 0, the EA is the sum of the contents of GPR `RA` and `D`, a 16-bit, signed two’s complement integer sign-extended to 32 bits. If GPR `RA` is 0, then the EA is `D`.

The `lfd` instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

**Parameters**

- **FRT**: Specifies target general-purpose register where result of the operation is stored.
- **D**: 16-bit, signed two’s complement integer sign-extended to 32 bits for the EA calculation.
- **RA**: Specifies source general-purpose register for the EA calculation.

**Examples**

The following code loads a doubleword from memory into FPR 6:

```
.csect data[rv]
storage: .double 0x1
# Assume GPR 5 contains the address of csect data[rv].
.csect text[pr]
1fd 6,storage(5)
# FPR 6 now contains 0x3FF0 0000 0000 0000.
```

**Related Information**

- [Floating-Point Processor](#)
- [Floating-Point Load and Store Instructions](#)

---

### Ifdu (Load Floating-Point Double with Update) Instruction

**Purpose**

Loads a doubleword of data from a specified location in memory into a floating-point register and possibly places the specified address in a general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>51</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>
Description

The `lfdu` instruction loads a doubleword in storage from a specified location in memory addressed by the effective address (EA) into the target floating-point register (FPR) `FRT`.

If `RA` is not 0, the EA is the sum of the contents of GPR `RA` and `D`, a 16-bit, signed two's complement integer sign-extended to 32 bits. If `RA` is 0, then the effective address (EA) is `D`.

If `RA` does not equal 0, and the storage access does not cause an Alignment interrupt or a Data Storage interrupt, then the effective address is stored in GPR `RA`.

The `lfdu` instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

- **FRT**: Specifies target general-purpose register where result of operation is stored.
- **D**: Specifies a 16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
- **RA**: Specifies source general-purpose register for EA calculation and possible address update.

Examples

The following code loads a doubleword from memory into FPR 6 and stores the address in GPR 5:

```
.csect data[rw]
storage: .double 0x1
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
lfdu 6,storage(5)
# FPR 6 now contains 0x3FF0 0000 0000 0000.
# GPR 5 now contains the storage address.
```

Related Information

- Floating-Point Processor
- Floating-Point Load and Store Instructions

**Ifdux (Load Floating-Point Double with Update Indexed) Instruction**

Purpose

 Loads a doubleword of data from a specified location in memory into a floating-point register and possibly places the specified address in a general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>631</td>
</tr>
<tr>
<td>Bits</td>
<td>Value</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

**Lfdux** \( \text{FRT, RA, RB} \)

**Description**

The **Lfdux** instruction loads a doubleword in storage from a specified location in memory addressed by the effective address (EA) into the target floating-point register (FPR) \( \text{FRT} \).

If \( \text{RA} \) is not 0, the EA is the sum of the contents of general-purpose register (GPR) \( \text{RA} \) and GPR \( \text{RB} \). If \( \text{RA} \) is 0, then the EA is the contents of \( \text{RB} \).

If \( \text{RA} \) does not equal 0, and the storage access does not cause an Alignment interrupt or a Data Storage interrupt, then the EA is stored in GPR \( \text{RA} \).

The **Lfdux** instruction has one syntax form and does not affect the Floating-Point Status and Control Register.

**Parameters**

- **FRT** Specifies target general-purpose register where result of operation is stored.
- **RA** Specifies source general-purpose register for EA calculation.
- **RB** Specifies source general-purpose register for EA calculation.

**Examples**

The following code loads a doubleword from memory into FPR 6 and stores the address in GPR 5:

```assembly
.csect data[rw]
storage: .double 0x1
# Assume GPR 5 contains the address of .csect data[rw].
# Assume GPR 4 contains the displacement of storage relative
# to .csect data[rw].
.csect text[pr]
lfdux 6,5,4
# FPR 6 now contains 0x3FF0 0000 0000 0000.
# GPR 5 now contains the storage address.
```

**Related Information**

- Floating-Point Processor
- Floating-Point Load and Store Instructions

**Lfdux (Load Floating-Point Double-Indexed) Instruction**

**Purpose**

Loads a doubleword of data from a specified location in memory into a floating-point register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
</tbody>
</table>
Description
The **lfdx** instruction loads a doubleword in storage from a specified location in memory addressed by the effective address (EA) into the target floating-point register (FPR) **FRT**.

If **RA** is not 0, the EA is the sum of the contents of general-purpose register (GPR) **RA** and GPR **RB**. If **RA** is 0, then the EA is the contents of GPR **RB**.

The **lfdx** instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

- **FRT** Specifies target floating-point register where data is stored.
- **RA** Specifies source general-purpose register for EA calculation.
- **RB** Specifies source general-purpose register for EA calculation.

Examples
The following code loads a doubleword from memory into FPR 6:

```assembly
storage: .double 0x1
.
.
# Assume GPR 4 contains the storage address.
lfdx 6,0,4
# FPR 6 now contains 0x3FF0 0000 0000 0000.
```

Related Information
- [Floating-Point Processor](#)
- [Floating-Point Load and Store Instructions](#)

**lfq (Load Floating-Point Quad) Instruction**

Purpose
Loads two double-precision values into floating-point registers.

**Note**: The **lfq** instruction is supported only in the POWER2 implementation of the POWER family architecture.
Syntax

<table>
<thead>
<tr>
<th></th>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0-5</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td></td>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td></td>
<td>16-29</td>
<td>DS</td>
</tr>
<tr>
<td></td>
<td>30-31</td>
<td>00</td>
</tr>
</tbody>
</table>

POWER2
lfq  FRT  DS  (RA)

Description

The `lfq` instruction loads the two doublewords from the location in memory specified by the effective address (EA) into two floating-point registers (FPR).

`DS` is sign-extended to 30 bits and concatenated on the right with b'00' to form the offset value. If general-purpose register (GPR) `RA` is 0, the offset value is the EA. If GPR `RA` is not 0, the offset value is added to GPR `RA` to generate the EA. The doubleword at the EA is loaded into FPR `FRT`. If `FRT` is 31, the doubleword at EA+8 is loaded into FPR 0; otherwise, it is loaded into `FRT+1`.

The `lfq` instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

- **FRT** Specifies the first of two target floating-point registers.
- **DS** Specifies a 14-bit field used as an immediate value for the EA calculation.
- **RA** Specifies one source general-purpose register for the EA calculation.

Examples

The following code copies two double-precision floating-point values from one place in memory to a second place in memory:

```
# Assume GPR 3 contains the address of the first source floating-point value.
# Assume GPR 4 contains the address of the target location.
lfq  7,0(3)  # Load first two values into FPRs 7 and 8.
stfq 7,0(4)  # Store the two doublewords at the new location.
```

Related Information

- [Floating-Point Processor](#)
- [Floating-Point Load and Store Instructions](#)

Ifqu (Load Floating-Point Quad with Update) Instruction

Purpose

Loads two double-precision values into floating-point registers and updates the address base.
Note: The \texttt{lfqu} instruction is supported only in the POWER2 implementation of the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>57</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-29</td>
<td>DS</td>
</tr>
<tr>
<td>30-31</td>
<td>00</td>
</tr>
</tbody>
</table>

**POWER2**

\texttt{lfqu} \hspace{1cm} \texttt{FRT DS RA}

**Description**

The \texttt{lfqu} instruction loads the two doublewords from the location in memory specified by the effective address (EA) into two floating-point registers (FPR).

\textit{DS} is sign-extended to 30 bits and concatenated on the right with b'00' to form the offset value. If general-purpose register GPR \textit{RA} is 0, the offset value is the EA. If GPR \textit{RA} is not 0, the offset value is added to GPR \textit{RA} to generate the EA. The doubleword at the EA is loaded into FPR \textit{FRT}. If \textit{FRT} is 31, the doubleword at EA+8 is loaded into FPR 0; otherwise, it is loaded into \textit{FRT+1}.

If GPR \textit{RA} is not 0, the EA is placed into GPR \textit{RA}.

The \texttt{lfqu} instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

**Parameters**

\textit{FRT} \hspace{0.5cm} Specifies the first of two target floating-point register.
\textit{DS} \hspace{0.5cm} Specifies a 14-bit field used as an immediate value for the EA calculation.
\textit{RA} \hspace{0.5cm} Specifies one source general-purpose register for EA calculation and the target register for the EA update.

**Examples**

The following code calculates the sum of six double-precision floating-point values that are located in consecutive doublewords in memory:

```
# Assume GPR 3 contains the address of the first
# floating-point value.
# Assume GPR 4 contains the address of the target location.
lfq    7,0(3)  # Load first two values into FPRs 7 and 8.
lfqu   9,16(3) # Load next two values into FPRs 9 and 10
             # and update base address in GPR 3.
fadd   6,7,8   # Add first two values.
lfq    7,16(3) # Load next two values into FPRs 7 and 8.
fadd   6,6,9   # Add third value.
fadd   6,6,10  # Add fourth value.
fadd   6,6,7   # Add fifth value.
fadd   6,6,8   # Add sixth value.
stfqx   7,0,4   # Store the two doublewords at the new
                # location.
```
Related Information
Floating-Point Processor.
Floating-Point Load and Store Instructions.

Ifqux (Load Floating-Point Quad with Update Indexed) Instruction

Purpose
Loads two double-precision values into floating-point registers and updates the address base.

Note: The Ifqux instruction is supported only in the POWER2 implementation of the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>823</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER2
Ifqux FRT RA RB

Description
The Ifqux instruction loads the two doublewords from the location in memory specified by the effective address (EA) into two floating-point registers (FPR).

If general-purpose register (GPR) RA is not 0, the EA is the sum of the contents of GPR RA and GPR RB. If GPR RA is 0, the EA is the contents of GPR RB. The doubleword at the EA is loaded into FPR FRT. If FRT is 31, the doubleword at EA+8 is loaded into FPR 0; otherwise, it is loaded into FRT+1.

If GPR RA is not 0, the EA is placed into GPR RA.

The Ifqux instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters
- FRT Specifies the first of two target floating-point registers.
- RA Specifies the first source general-purpose register for the EA calculation and the target register for the EA update.
- RB Specifies the second source general-purpose register for the EA calculation.

Examples
The following code calculates the sum of three double-precision, floating-point, two-dimensional coordinates:
Assume the two-dimensional coordinates are contained in a linked list with elements of the form:

- `list_element`
  - `.double` Floating-point value of X.
  - `.double` Floating-point value of Y.
  - `.next_elem` Offset to next element;
    - # from X(n) to X(n+1).
  - # Assume GPR 3 contains the address of the first list element.
  - # Assume GPR 4 contains the address where the resultant sums will be stored.

Get first pair of X_Y values.

- `lwz` 5,16(3) Get the offset to second element.
- `lfqu` 9,3,5 Get second pair of X_Y values.
- `lwz` 5,16(3) Get the offset to third element.
- `fadd` 7,7,9 Add first two X values.
- `fadd` 8,8,10 Add first two Y values.
- `lfqu` 9,3,5 Get third pair of X_Y values.
- `fadd` 7,7,9 Add third X value to sum.
- `fadd` 8,8,10 Add third Y value to sum.
- `stfq` 7,8,4 Store the two doubleword results.

**Related Information**

- [Floating-Point Processor](#)
- [Floating-Point Load and Store Instructions](#)

### Ifqx (Load Floating-Point Quad Indexed) Instruction

**Purpose**

Loads two double-precision values into floating-point registers.

**Note:** The `Ifqx` instruction is supported only in the POWER2 implementation of the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>791</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**POWER2**

- `Ifqx` [FRT] [RA] [RB]

**Description**

The `Ifqx` instruction loads the two doublewords from the location in memory specified by the effective address (EA) into two floating-point registers (FPR).
If general-purpose register (GPR) RA is not 0, the EA is the sum of the contents of GPR RA and GPR RB. If GPR RA is 0, the EA is the contents of GPR RB. The doubleword at the EA is loaded into FPR FRT. If FRT is 31, the doubleword at EA+8 is loaded into FPR 0; otherwise, it is loaded into FRT+1.

The Ifqx instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

**Parameters**

FRT  Specifies the first of two target floating-point registers.
RA   Specifies one source general-purpose register for the EA calculation.
RB   Specifies the second source general-purpose register for the EA calculation.

**Examples**

The following code calculates the sum of two double-precision, floating-point values that are located in consecutive doublewords in memory:

```plaintext
# Assume GPR 3 contains the address of the first floating-point value.
# Assume GPR 4 contains the address of the target location.
lfqx 7,0,3  # Load values into FPRs 7 and 8.
fadd 7,7,8  # Add the two values.
stfdx 7,0,4 # Store the doubleword result.
```

**Related Information**

Floating-Point Processor  
Floating-Point Load and Store Instructions  

**Ifs (Load Floating-Point Single) Instruction**

**Purpose**

Loads a floating-point, single-precision number that has been converted to a floating-point, double-precision number into a floating-point register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>48</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

**Description**

The Ifs instruction converts a floating-point, single-precision word in storage addressed by the effective address (EA) to a floating-point, double-precision word and loads the result into floating-point register (FPR) FRT.
If RA is not 0, the EA is the sum of the contents of GPR RA and D, a 16-bit, signed two’s complement integer sign-extended to 32 bits. If RA is 0, then the EA is D.

The lfs instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRT</td>
<td>Specifies target floating-point register where data is stored.</td>
</tr>
<tr>
<td>D</td>
<td>16-bit, signed two’s complement integer sign-extended to 32 bits for EA calculation.</td>
</tr>
<tr>
<td>RA</td>
<td>Specifies source general-purpose register for EA calculation.</td>
</tr>
</tbody>
</table>

Examples

The following code loads the single-precision contents of storage into FPR 6:

```
csect data[rw]
storage: .float 0x1
# Assume GPR 5 contains the address csect data[rw].
csect text[pr]
lfs 6,storage(5)
# FPR 6 now contains 0x3FF0 0000 0000 0000.
```

Related Information

Floating-Point Processor.

Floating-Point Load and Store Instructions.

**lfsu (Load Floating-Point Single with Update) Instruction**

Purpose

Loads a floating-point, single-precision number that has been converted to a floating-point, double-precision number into a floating-point register and possibly places the effective address in a general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>49</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

**lfsu** 

| FRT | D | RA |

Description

The **lfsu** instruction converts a floating-point, single-precision word in storage addressed by the effective address (EA) to floating-point, double-precision word and loads the result into floating-point register (FPR) FRT.

If RA is not 0, the EA is the sum of the contents of general-purpose register (GPR) RA and D, a 16-bit signed two’s complement integer sign extended to 32 bits. If RA is 0, then the EA is D.
If RA does not equal 0 and the storage access does not cause an Alignment interrupt or a Data Storage interrupt, then the EA is stored in GPR RA.

The lfsu instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

FRT Specifies target floating-point register where data is stored.
D 16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
RA Specifies source general-purpose register for EA calculation and possible address update.

Examples

The following code loads the single-precision contents of storage, which is converted to double precision, into FPR 6 and stores the effective address in GPR 5:

```
csect data[rw]
  storage: .float 0x1
.csect text[pr]
  # Assume GPR 5 contains the storage address.
  lfsu 6,0(5)
  # FPR 6 now contains 0x3FF0 0000 0000 0000.
  # GPR 5 now contains the storage address.
```

Related Information

Floating-Point Processor

Floating-Point Load and Store Instructions

Ifsux (Load Floating-Point Single with Update Indexed) Instruction

Purpose

Loads a floating-point, single-precision number that has been converted to a floating-point, double-precision number into a floating-point register and possibly places the effective address in a general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>567</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

Ifsux  

FRT RA RB
**Description**

The **lfsux** instruction converts a floating-point, single-precision word in storage addressed by the effective address (EA) to floating-point, double-precision word and loads the result into floating-point register (FPR) **FRT**.

If general-purpose register (GPR) **RA** is not 0, the EA is the sum of the contents of GPR **RA** and GPR **RB**. If **RA** is 0, then the EA is the contents of GPR **RB**.

If GPR **RA** does not equal 0 and the storage access does not cause an Alignment interrupt or a Data Storage interrupt, then the EA is stored in GPR **RA**.

The **lfsux** instruction has one syntax form and does not affect the Floating-Point Status Control Register.

**Parameters**

- **FRT** Specifies target floating-point register where data is stored.
- **RA** Specifies source general-purpose register for EA calculation and possible address update.
- **RB** Specifies source general-purpose register for EA calculation.

**Examples**

The following code loads the single-precision contents of storage into FPR 6 and stores the effective address in GPR 5:

```assembly
.csect data[rw]
storage: .float 0x1

# Assume GPR 4 contains the address of csect data[rw].
# Assume GPR 5 contains the displacement of storage
# relative to .csect data[rw].
.csect text[pr]
lfsux 6,5,4
# FPR 6 now contains 0x3FF0 0000 0000 0000.
# GPR 5 now contains the storage address.
```

**Related Information**

- [Floating-Point Processor](#)
- [Floating-Point Load and Store Instructions](#)

### Ifsx (Load Floating-Point Single Indexed) Instruction

**Purpose**

Loads a floating-point, single-precision number that has been converted to a floating-point, double-precision number into a floating-point register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>535</td>
</tr>
</tbody>
</table>
Description
The lfsx instruction converts a floating-point, single-precision word in storage addressed by the effective address (EA) to floating-point, double-precision word and loads the result into floating-point register (FPR) FRT.

If general-purpose register (GPR) RA is not 0, the EA is the sum of the contents of GPR RA and GPR RB. If RA is 0, then the EA is the contents of GPR RB.

The lfsx instruction has one syntax form and does not affect the Floating-Point Status and Control Register.

Parameters

\[ FRT \] Specifies target floating-point register where data is stored.
\[ RA \] Specifies source general-purpose register for EA calculation.
\[ RB \] Specifies source general-purpose register for EA calculation.

Examples
The following code loads the single-precision contents of storage into FPR 6:

```
storage: .float 0x1.
    # Assume GPR 4 contains the address of storage.
    lfsx 6,0,4
    # FPR 6 now contains 0x3FF0 0000 0000 0000.
```

Related Information
Floating-Point Processor
Floating-Point Load and Store Instructions

Iha (Load Half Algebraic) Instruction

Purpose
Loads a halfword of data from a specified location in memory into a general-purpose register and copies bit 0 of the halfword into the remaining 16 bits of the general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>42</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>
Description
The lha instruction loads a halfword of data from a specified location in memory, addressed by the effective address (EA), into bits 16-31 of the target general-purpose register (GPR) RT and copies bit 0 of the halfword into bits 0-15 of GPR RT.

If GPR RA is not 0, the EA is the sum of the contents of GPR RA and D, a 16-bit signed two's complement integer sign extended to 32 bits. If GPR RA is 0, then the EA is D.

The lha instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters
- RT Specifies target general-purpose register where result of operation is stored.
- D 16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
- RA Specifies source general-purpose register for EA calculation.

Examples
The following code loads a halfword of data into bits 16-31 of GPR 6 and copies bit 0 of the halfword into bits 0-15 of GPR 6:

```assembly
.csect data[rw]
storage: .short 0xffff
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
lha 6,storage(5)
# GPR 6 now contains 0xffff ffff.
```

Related Information
- Fixed-Point Processor
- Fixed-Point Load and Store Instructions

Ihau (Load Half Algebraic with Update) Instruction

Purpose
Loads a halfword of data from a specified location in memory into a general-purpose register, copies bit 0 of the halfword into the remaining 16 bits of the general-purpose register, and possibly places the address in another general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>43</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>
Description

The \texttt{lhau} instruction loads a halfword of data from a specified location in memory, addressed by the effective address (EA), into bits 16-31 of the target general-purpose register (GPR) $RT$ and copies bit 0 of the halfword into bits 0-15 of GPR $RT$.

If GPR $RA$ is not 0, the EA is the sum of the contents of GPR $RA$ and $D$, a 16-bit, signed two’s complement integer sign-extended to 32 bits. If GPR $RA$ is 0, then the EA is $D$.

If $RA$ does not equal $RT$ and $RA$ does not equal 0, and the storage access does not cause an Alignment interrupt or a Data Storage interrupt, then the EA is placed into GPR $RA$.

The \texttt{lhau} instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

$RT$ \hspace{1cm} Specifies target general-purpose register where result of operation is stored.

$D$ \hspace{1cm} 16-bit, signed two’s complement integer sign-extended to 32 bits for EA calculation.

$RA$ \hspace{1cm} Specifies source general-purpose register for EA calculation and possible address update.

Examples

The following code loads a halfword of data into bits 16-31 of GPR 6, copies bit 0 of the halfword into bits 0-15 of GPR 6, and stores the effective address in GPR 5:

```
csect data[rw]
storage: .short 0xffff
# Assume GPR 5 contains the address of csect data[rw].
csect text[pr]
lhau 6,storage(5)
# GPR 6 now contains 0xffff ffff.
# GPR 5 now contains the address of storage.
```

Related Information

Fixed-Point Processor.

Fixed-Point Load and Store with Update Instructions.

\textbf{lhau} (Load Half Algebraic with Update Indexed) Instruction

Purpose

Loads a halfword of data from a specified location in memory into a general-purpose register, copies bit 0 of the halfword into the remaining 16 bits of the general-purpose register, and possibly places the address in another general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>$RT$</td>
</tr>
<tr>
<td>11-15</td>
<td>$RA$</td>
</tr>
</tbody>
</table>
### Description

The `lhaux` instruction loads a halfword of data from a specified location in memory addressed by the effective address (EA) into bits 16-31 of the target general-purpose register (GPR) $RT$ and copies bit 0 of the halfword into bits 0-15 of GPR $RT$.

If GPR $RA$ is not 0, the EA is the sum of the contents of GPR $RA$ and GPR $RB$. If GPR $RA$ is 0, then the EA is the contents of GPR $RB$.

If $RA$ does not equal $RT$ and $RA$ does not equal 0, and the storage access does not cause an Alignment interrupt or a Data Storage interrupt, then the EA is placed into GPR $RA$.

The `lhaux` instruction has one syntax form and does not affect the Fixed-Point Exception Register.

### Parameters

- **$RT$**: Specifies target general-purpose register where result of operation is stored.
- **$RA$**: Specifies first source general-purpose register for EA calculation and possible address update.
- **$RB$**: Specifies second source general-purpose register for EA calculation.

### Examples

The following code loads a halfword of data into bits 16-31 of GPR 6, copies bit 0 of the halfword into bits 0-15 of GPR 6, and stores the effective address in GPR 5:

```
.csect data[rw]
storage: .short 0xffff
# Assume GPR 5 contains the address of csect data[rw].
# Assume GPR 4 contains the displacement of storage relative to data[rw].
.csect text[pr]
lhaux 6,5,4
# GPR 6 now contains 0xffff ffff.
# GPR 5 now contains the storage address.
```

### Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Load and Store with Update Instructions](#)

---

**Ihax (Load Half Algebraic Indexed) Instruction**

### Purpose

Loads a halfword of data from a specified location in memory into a general-purpose register and copies bit 0 of the halfword into the remaining 16 bits of the general-purpose register.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>343</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

Ihax \[RT\ RA\ RB\]

Description

The *Ihax* instruction loads a halfword of data from a specified location in memory, addressed by the effective address (EA), into bits 16-31 of the target general-purpose register (GPR) *RT* and copies bit 0 of the halfword into bits 0-15 of GPR *RT*.

If GPR *RA* is not 0, the EA is the sum of the contents of GPR *RA* and GPR *RB*. If GPR *RA* is 0, then the EA is the contents of GPR *RB*.

The *Ihax* instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters

*RT* Specifies target general-purpose register where result of operation is stored.

*RA* Specifies source general-purpose register for EA calculation.

*RB* Specifies source general-purpose register for EA calculation.

Examples

The following code loads a halfword of data into bits 16-31 of GPR 6 and copies bit 0 of the halfword into bits 0-15 of GPR 6:

```
csect data[rw]
.short 0x1
# Assume GPR 5 contains the address of csect data[rw].
# Assume GPR 4 contains the displacement of the halfword relative to data[rw].
csect text[pr]
Ihax 6,5,4
# GPR 6 now contains 0x0000 0001.
```

Related Information

[Fixed-Point Processor](#).

[Fixed-Point Load and Store Instructions](#).

**Ihbrx (Load Half Byte-Reverse Indexed) Instruction**

**Purpose**

Loads a byte-reversed halfword of data from a specified location in memory into a general-purpose register and sets the remaining 16 bits of the general-purpose register to zero.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>790</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

**lhbrx**  
**RT RA RB**

**Description**

The **lhbrx** instruction loads bits 00-07 and bits 08-15 of the halfword in storage addressed by the effective address (EA) into bits 24-31 and bits 16-23 of general-purpose register (GPR) **RT**, and sets bits 00-15 of GPR **RT** to 0.

If GPR **RA** is not 0, the EA is the sum of the contents of GPR **RA** and GPR **RB**. If GPR **RA** is 0, then the EA is the contents of GPR **RB**.

The **lhbrx** instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Parameters**

- **RT** Specifies target general-purpose register where result of operation is stored.
- **RA** Specifies source general-purpose register for EA calculation.
- **RB** Specifies source general-purpose register for EA calculation.

**Examples**

The following code loads bits 00-07 and bits 08-15 of the halfword in storage into bits 24-31 and bits 16-23 of GPR 6, and sets bits 00-15 of GPR 6 to 0:

```
csect data[rw]
.short 0x7654
# Assume GPR 4 contains the address of csect data[rw].
# Assume GPR 5 contains the displacement relative to data[rw].
csect text[pr]
lhbrx 6,5,4
# GPR 6 now contains 0x0000 5476.
```

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Load and Store Instructions](#)
Ihz (Load Half and Zero) Instruction

Purpose
Loads a halfword of data from a specified location in memory into a general-purpose register and sets the remaining 16 bits to 0.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>40</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

```lhz RT, D (RA)```

Description
The `Ihz` instruction loads a halfword of data from a specified location in memory, addressed by the effective address (EA), into bits 16-31 of the target general-purpose register (GPR) `RT` and sets bits 0-15 of GPR `RT` to 0.

If GPR `RA` is not 0, the EA is the sum of the contents of GPR `RA` and `D`, a 16-bit, signed two’s complement integer sign-extended to 32 bits. If GPR `RA` is 0, then the EA is `D`.

The `Ihz` instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

- `RT` Specifies target general-purpose register where result of operation is stored.
- `D` 16-bit, signed two’s complement integer sign-extended to 32 bits for EA calculation.
- `RA` Specifies source general-purpose register for EA calculation.

Examples
The following code loads a halfword of data into bits 16-31 of GPR 6 and sets bits 0-15 of GPR 6 to 0:

```c
.csect data[rw]
storage: .short 0xffff
# Assume GPR 4 holds the address of csect data[rw].
.csect text[pr]
lhz 6,storage(4)
# GPR 6 now holds 0x0000 ffff.
```

Related Information
- Fixed-Point Processor
- Fixed-Point Load and Store Instructions
Ihzu (Load Half and Zero with Update) Instruction

Purpose
Loads a halfword of data from a specified location in memory into a general-purpose register, sets the remaining 16 bits of the general-purpose register to 0, and possibly places the address in another general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>41</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

Ihzu \( RT[0][RA] \)

Description
The Ihzu instruction loads a halfword of data from a specified location in memory, addressed by the effective address (EA), into bits 16-31 of the target general-purpose register (GPR) \( RT \) and sets bits 0-15 of GPR \( RT \) to 0.

If GPR RA is not 0, the EA is the sum of the contents of GPR RA and \( D \), a 16-bit, signed two’s complement integer sign-extended to 32 bits. If GPR RA is 0, then the EA is \( D \).

If \( RA \) does not equal \( RT \) and \( RA \) does not equal 0, and the storage access does not cause an Alignment interrupt or a Data Storage interrupt, then the EA is placed into GPR RA.

The Ihzu instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

- \( RT \) Specifies target general-purpose register where result of operation is stored.
- \( D \) 16-bit, signed two’s complement integer sign-extended to 32 bits for EA calculation.
- \( RA \) Specifies source general-purpose register for EA calculation and possible address update.

Examples
The following code loads a halfword of data into bits 16-31 of GPR 6, sets bits 0-15 of GPR 6 to 0, and stores the effective address in GPR 4:

```assembly
.csect data[rw]
.short 0xffff
  # Assume GPR 4 contains the address of csect data[rw].
.csect text[pr]
Ihzu 6,0(4)
  # GPR 6 now contains 0x0000 ffff.
```
Related Information
- Fixed-Point Processor
- Fixed-Point Load and Store with Update Instructions

Ihzux (Load Half and Zero with Update Indexed) Instruction

Purpose
Loads a halfword of data from a specified location in memory into a general-purpose register, sets the remaining 16 bits of the general-purpose register to 0, and possibly places the address in another general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>331</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

**Description**
The Ihzux instruction loads a halfword of data from a specified location in memory, addressed by the effective address (EA), into bits 16-31 of the target general-purpose register (GPR) $RT$ and sets bits 0-15 of GPR $RT$ to 0.

If GPR $RA$ is not 0, the EA is the sum of the contents of GPR $RA$ and GPR $RB$. If GPR $RA$ is 0, then the EA is the contents of GPR $RB$.

If $RA$ does not equal $RT$ and $RA$ does not equal 0, and the storage access does not cause an Alignment interrupt or a Data Storage interrupt, then the EA is placed into GPR $RA$.

The Ihzux instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters
- $RT$ Specifies target general-purpose register where result of operation is stored.
- $RA$ Specifies source general-purpose register for EA calculation and possible address update.
- $RB$ Specifies source general-purpose register for EA calculation.

Examples
The following code loads a halfword of data into bits 16-31 of GPR 6, sets bits 0-15 of GPR 6 to 0, and stores the effective address in GPR 5:

```assembly
.csect data[rw]
storage: .short 0xffff
# Assume GPR 5 contains the address of csect data[rw].
```
Assume GPR 4 contains the displacement of storage relative to data[rw].
.csect text[pr]
lhzux 6,5,4
# GPR 6 now contains 0x0000 ffff.
# GPR 5 now contains the storage address.

Related Information
Fixed-Point Processor.
Fixed-Point Load and Store with Update Instructions.

Ihzx (Load Half and Zero Indexed) Instruction

Purpose
Loads a halfword of data from a specified location in memory into a general-purpose register and sets the remaining 16 bits of the general-purpose register to 0.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>279</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

Description
The Ihzx instruction loads a halfword of data from a specified location in memory, addressed by the effective address (EA), into bits 16-31 of the target general-purpose register (GPR) RT and sets bits 0-15 of GPR RT to 0.

If GPR RA is not 0, the EA is the sum of the contents of GPR RA and GPR RB. If GPR RA is 0, then the EA is the contents of GPR RB.

The Ihzx instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

- **RT**: Specifies target general-purpose register where result of operation is stored.
- **RA**: Specifies source general-purpose register for EA calculation.
- **RB**: Specifies source general-purpose register for EA calculation.

Examples
The following code loads a halfword of data into bits 16-31 of GPR 6 and sets bits 0-15 of GPR 6 to 0:
Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Load and Store Instructions](#)

### Imw or Im (Load Multiple Word) Instruction

#### Purpose

Loads consecutive words at a specified location into more than one general-purpose register.

#### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>46</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

**PowerPC**

```plaintext
Imw  RT  D  RA
```

**POWER family**

```plaintext
Im   RT  D  RA
```

#### Description

The **Imw** and **Im** instructions load \( N \) consecutive words starting at the calculated effective address (EA) into a number of general-purpose registers (GPR), starting at GPR \( RT \) and filling all GPRs through GPR 31. \( N \) is equal to 32-\( RT \) field, the total number of consecutive words that are placed in consecutive registers.

If GPR \( RA \) is not 0, the EA is the sum of the contents of GPR \( RA \) and \( D \). If GPR \( RA \) is 0, then the EA is \( D \).

Consider the following when using the PowerPC instruction **Imw**:

- If GPR \( RA \) or GPR \( RB \) is in the range of registers to be loaded or \( RT = RA = 0 \), the results are boundedly undefined.
- The EA must be a multiple of 4. If it is not, the system alignment error handler may be invoked or the results may be boundedly undefined.

For the POWER family instruction **Im**, if GPR \( RA \) is not equal to 0 and GPR \( RA \) is in the range to be loaded, then GPR \( RA \) is not written to. The data that would have normally been written into \( RA \) is discarded and the operation continues normally.
The `lmw` and `lm` instructions have one syntax and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Note:** The `lmw` and `lm` instructions are interruptible due to a data storage interrupt. When such an interrupt occurs, the instruction should be restarted from the beginning.

**Parameters**

- **RT**  
  Specifies starting target general-purpose register for operation.
- **D**  
  Specifies a 16-bit signed two's complement integer sign extended to 32 bits for EA calculation
- **RA**  
  Specifies source general-purpose register for EA calculation.

**Examples**

The following code loads data into GPR 29 and GPR 31:

```assembly
.csect data[rw]
.long 0x8971
.long -1
.long 0x7ffe c100
# Assume GPR 30 contains the address of csect data[rw].
.csect text[pr]
1mw 29,0(30)
# GPR 29 now contains 0x0000 8971.  
# GPR 30 now contains the address of csect data[rw].
# GPR 31 now contains 0x7ffe c100.
```

**Related Information**

- [Fixed-Point Processor](#).
- [Fixed-Point Load and Store Instructions](#).

---

### Iq (Load Quad Word) Instruction

**Purpose**

Load a quad-word of data into the specified general purpose register.

**Note:** This instruction should only be used on 64-bit PowerPC processors running a 64-bit application.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>56</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-27</td>
<td>DQ</td>
</tr>
<tr>
<td>28-31</td>
<td>PT</td>
</tr>
</tbody>
</table>

**PowerPC 64**

[“RT” on page 278](#)  [“DQ” on page 278](#)  [“RA” on page 278](#)  [“PT” on page 278](#)
Description
The **lq** instruction loads a quad word in storage from a specified location in memory addressed by the effective address (EA) into the target general-purpose registers (GPRs) **RT** and **RT+1**.

If GPR **RA** is not 0, the EA is the sum of the contents of GPR **RA** and **DQ**, a 12-bit, signed two’s complement integer, which is concatenated on the right by 0b0000 and sign-extended to 64 bits. If GPR **RA** is 0, then the EA is **DQ**.

Parameters
- **RT**: Specifies target general-purpose register where result of operation is stored. If RT is odd, the instruction form is invalid.
- **DQ**: Specifies a 12-bit, signed two’s complement integer, concatenated on the right with 0b0000, and sign-extended to 64 bits for EA calculation.
- **RA**: Specifies source general-purpose register for EA calculation.
- **PT**: Specifies a 4-bit unsigned immediate value.

Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Related Information
“Fixed-Point Processor” on page 21.
“Fixed-Point Load and Store Instructions” on page 21.

**lscbx (Load String and Compare Byte Indexed) Instruction**

Purpose
Loads consecutive bytes in storage into consecutive registers.

**Note**: The **lscbx** instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>277</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family
**lscbx**
**lscbx.**
Description
The **lscbx** instruction loads *N* consecutive bytes addressed by effective address (EA) into general-purpose register (GPR) *RT*, starting with the leftmost byte in register *RT*, through *RT* + *NR* - 1, and wrapping around back through GPR 0, if required, until either a byte match is found with XER16-23 or *N* bytes have been loaded. If a byte match is found, then that byte is also loaded.

If GPR *RA* is not 0, the EA is the sum of the contents of GPR *RA* and the address stored in GPR *RB*. If *RA* is 0, then EA is the contents of GPR *RB*.

Consider the following when using the **lscbx** instruction:
- XER(16-23) contains the byte to be compared.
- XER(25-31) contains the byte count before the instruction is invoked and the number of bytes loaded after the instruction has completed.
- If XER(25-31) = 0, GPR *RT* is not altered.
- *N* is XER(25-31), which is the number of bytes to load.
- *NR* is ceiling(*N*/4), which is the total number of registers required to contain the consecutive bytes.

Bytes are always loaded left to right in the register. In the case when a match was found before *N* bytes were loaded, the contents of the rightmost bytes not loaded from that register and the contents of all succeeding registers up to and including register *RT* + *NR* - 1 are undefined. Also, no reference is made to storage after the matched byte is found. In the case when a match was not found, the contents of the rightmost bytes not loaded from register *RT* + *NR* - 1 are undefined.

If GPR *RA* is not 0 and GPRs *RA* and *RB* are in the range to be loaded, then GPRs *RA* and *RB* are not written to. The data that would have been written into them is discarded, and the operation continues normally. If the byte in XER(16-23) compares with any of the 4 bytes that would have been loaded into GPR *RA* or *RB*, but are being discarded for restartability, the EQ bit in the Condition Register and the count returned in XER(25-31) are undefined. The Multiply Quotient (MQ) Register is not affected by this operation.

The **lscbx** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>lscbx</td>
<td>None</td>
<td>XER(25-31) = # of bytes loaded</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>lscbx.</td>
<td>None</td>
<td>XER(25-31) = # of bytes loaded</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the **lscbx** instruction place the number of bytes loaded into Fixed-Point Exception Register (XER) bits 25-31. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0. If Rc = 1 and XER(25-31) = 0, then Condition Register Field 0 is undefined. If Rc = 1 and XER(25-31) <> 0, then Condition Register Field 0 is set as follows:

\[
\text{LT, GT, EQ, SO} = \text{b'00'} | \text{match} | \text{XER(SO)}
\]

**Note:** This instruction can be interrupted by a Data Storage interrupt. When such an interrupt occurs, the instruction is restarted from the beginning.
Parameters

- **RT** Specifies the starting target general-purpose register.
- **RA** Specifies source general-purpose register for EA calculation.
- **RB** Specifies source general-purpose register for EA calculation.

Examples

1. The following code loads consecutive bytes into GPRs 6, 7, and 8:
   ```
   .csect data[rw]
   string: "Hello, world"
   # Assume XER16-23 = 'a'.
   # Assume XER25-31 = 9.
   # Assume GPR 5 contains the address of csect data[rw].
   # Assume GPR 4 contains the displacement of string relative
   # to csect data[rw].
   .csect text[pr]
   lscbx 6,5,4
   # GPR 6 now contains 0x4865 6c6c.
   # GPR 7 now contains 0x6f2c 2077.
   # GPR 8 now contains 0xXXXX XXXX.
   ```

2. The following code loads consecutive bytes into GPRs 6, 7, and 8:
   ```
   # Assume XER16-23 = 'e'.
   # Assume XER25-31 = 9.
   # Assume GPR 5 contains the address of csect data[rw].
   # Assume GPR 4 contains the displacement of string relative
   # to csect data[rw].
   .csect text[pr]
   lscbx 6,5,4
   # GPR 6 now contains 0x4865 XXXX.
   # GPR 7 now contains 0xXXXX XXXX.
   # GPR 8 now contains 0xXXXX XXXX.
   # XER25-31 = 2.
   # CRF 0 now contains 0x2.
   ```

Related Information

- Fixed-Point Processor
- Fixed-Point String Instructions

**Iswi or Isi (Load String Word Immediate) Instruction**

**Purpose**

Loads consecutive bytes in storage from a specified location in memory into consecutive general-purpose registers.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>NB</td>
</tr>
<tr>
<td>21-30</td>
<td>597</td>
</tr>
</tbody>
</table>
Bits | Value
---|---
31 | /

### PowerPC

**lswi**  
\[ RT, RA, NB \]

### POWER family

**lsi**  
\[ RT, RA, NB \]

### Description

The **lswi** and **lsi** instructions load \( N \) consecutive bytes in storage addressed by the effective address (EA) into general-purpose register GPR \( RT \), starting with the leftmost byte, through GPR \( RT+NR-1 \), and wrapping around back through GPR 0, if required.

If GPR \( RA \) is not 0, the EA is the contents of GPR \( RA \). If GPR \( RA \) is 0, then the EA is 0.

Consider the following when using the **lswi** and **lsi** instructions:
- \( NB \) is the byte count.
- \( RT \) is the starting general-purpose register.
- \( N \) is \( NB \), which is the number of bytes to load. If \( NB \) is 0, then \( N \) is 32.
- \( NR \) is ceiling(N/4), which is the number of general-purpose registers to receive data.

For the PowerPC instruction **lswi**, if GPR \( RA \) is in the range of registers to be loaded or \( RT = RA = 0 \), the instruction form is invalid.

Consider the following when using the POWER family instruction **lsi**:
- If GPR \( RT + NR - 1 \) is only partially filled on the left, the rightmost bytes of that general-purpose register are set to 0.
- If GPR \( RA \) is in the range to be loaded, and if GPR \( RA \) is not equal to 0, then GPR \( RA \) is not written into by this instruction. The data that would have been written into it is discarded, and the operation continues normally.

The **lswi** and **lsi** instructions have one syntax form which does not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Note:** The **lswi** and **lsi** instructions can be interrupted by a Data Storage interrupt. When such an interrupt occurs, the instruction is restarted from the beginning.

### Parameters

- **RT**  
  Specifies starting general-purpose register of stored data.
- **RA**  
  Specifies general-purpose register for EA calculation.
- **NB**  
  Specifies byte count.

### Examples

The following code loads the bytes contained in a location in memory addressed by GPR 7 into GPR 6:
.csect data[rw]
.string "Hello, World"
# Assume GPR 7 contains the address of csect data[ rw].
.csect text[ pr]
lswi 6,7,0x6
# GPR 6 now contains 0x4865 6c6c.

Related Information
[Fixed-Point Processor].
[Fixed-Point String Instructions].

Iswx or Isx (Load String Word Indexed) Instruction

Purpose
Loads consecutive bytes in storage from a specified location in memory into consecutive general-purpose registers.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>533</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

PowerPC
lswx    \[RT\] \[RA\] \[RB\]

POWER family
lsx      \[RT\] \[RA\] \[RB\]

Description
The lswx and lsx instructions load \( N \) consecutive bytes in storage addressed by the effective address (EA) into general-purpose register (GPR) \( RT \), starting with the leftmost byte, through GPR \( RT + NR - 1 \), and wrapping around back through GPR 0 if required.

If GPR \( RA \) is not 0, the EA is the sum of the contents of GPR \( RA \) and the address stored in GPR \( RB \). If GPR \( RA \) is 0, then EA is the contents of GPR \( RB \).

Consider the following when using the lswx and lsx instructions:
- XER(25-31) contain the byte count.
- \( RT \) is the starting general-purpose register.
- \( N \) is XER(25-31), which is the number of bytes to load.
- \( NR \) is ceiling(N/4), which is the number of registers to receive data.
- If XER(25-31) = 0, general-purpose register \( RT \) is not altered.
For the PowerPC instruction **lswx**, if *RA* or *RB* is in the range of registers to be loaded or *RT = RA = 0*, the results are boundedly undefined.

Consider the following when using the POWER family instruction **lsx**:  
- If GPR *RT + NR - 1* is only partially filled on the left, the rightmost bytes of that general-purpose register are set to 0.  
- If GPRs *RA* and *RB* are in the range to be loaded, and if GPR *RA* is not equal to 0, then GPR *RA* and *RB* are not written into by this instruction. The data that would have been written into them is discarded, and the operation continues normally.

The **lswx** and **lsx** instructions have one syntax form which does not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Note:** The **lswx** and **lsx** instructions can be interrupted by a Data Storage interrupt. When such an interrupt occurs, the instruction is restarted from the beginning.

**Parameters**

<table>
<thead>
<tr>
<th>RT</th>
<th>Specifies starting general-purpose register of stored data.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>Specifies general-purpose register for EA calculation.</td>
</tr>
<tr>
<td>RB</td>
<td>Specifies general-purpose register for EA calculation.</td>
</tr>
</tbody>
</table>

**Examples**

The following code loads the bytes contained in a location in memory addressed by GPR 5 into GPR 6:

```
# Assume XER25-31 = 4.
csect data[rw]
storage: .string "Hello, world"
# Assume GPR 4 contains the displacement of storage
# relative to data[rw].
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
lswx 6,5,4
# GPR 6 now contains 0x4865 6c6c.
```

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point String Instructions](#)
- [Functional Differences for POWER family and PowerPC Instructions](#)

---

**Iwa (Load Word Algebraic) Instruction**

**Purpose**

Load a fullword of data from storage into the low-order 32 bits of the specified general purpose register. Sign extend the data into the high-order 32 bits of the register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>58</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
</tbody>
</table>
### Description

The fullword in storage located at the effective address (EA) is loaded into the low-order 32 bits of the target general purpose register (GRP) \( RT \). The value is then sign-extended to fill the high-order 32 bits of the register.

If GRP \( RA \) is not 0, the EA is the sum of the contents of GRP \( RA \) and \( B \); otherwise, the EA is equal to the contents of \( RB \).

### Parameters

- **\( RT \)** Specifies target general-purpose register where result of the operation is stored.
- **\( D \)** Specifies a 16-bit, signed two’s complement integer sign-extended to 32 bits for EA calculation.
- **\( RA \)** Specifies source general-purpose register for EA calculation.

### Implementation

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

### Iwarx (Load Word and Reserve Indexed) Instruction

#### Purpose

Used in conjunction with a subsequent [stwcx](#) instruction to emulate a read-modify-write operation on a specified memory location.

**Note:** The *Iwarx* instruction is supported only in the PowerPC architecture.

#### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>( RT )</td>
</tr>
<tr>
<td>11-15</td>
<td>( RA )</td>
</tr>
<tr>
<td>16-20</td>
<td>( RB )</td>
</tr>
<tr>
<td>21-30</td>
<td>20</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

PowerPC

*Iwarx*  \( RT \ RA \ RB \)
Description

The `lwarx` and `stwcx` instructions are primitive, or simple, instructions used to perform a read-modify-write operation to storage. If the store is performed, the use of the `lwarx` and `stwcx` instructions ensures that no other processor or mechanism has modified the target memory location between the time the `lwarx` instruction is executed and the time the `stwcx` instruction completes.

If general-purpose register (GPR) $RA = 0$, the effective address (EA) is the content of GPR $RB$. Otherwise, the EA is the sum of the content of GPR $RA$ plus the content of GPR $RB$.

The `lwarx` instruction loads the word from the location in storage specified by the EA into the target GPR $RT$. In addition, a reservation on the memory location is created for use by a subsequent `stwcx` instruction.

The `lwarx` instruction has one syntax form and does not affect the Fixed-Point Exception Register. If the EA is not a multiple of 4, the results are boundedly undefined.

Parameters

- $RT$ Specifies target general-purpose register where result of operation is stored.
- $RA$ Specifies source general-purpose register for EA calculation.
- $RB$ Specifies source general-purpose register for EA calculation.

Examples

1. The following code performs a “Fetch and Store” by atomically loading and replacing a word in storage:

   ```
   # Assume that GPR 4 contains the new value to be stored.
   # Assume that GPR 3 contains the address of the word
   # to be loaded and replaced.
   loop:   lwarx r5,0,r3  # Load and reserve
           stwcx. r4,0,r3  # Store new value if still
                       # reserved
           bne-  loop    # Loop if lost reservation
   # The new value is now in storage.
   # The old value is returned to GPR 4.
   ```

2. The following code performs a “Compare and Swap” by atomically comparing a value in a register with a word in storage:

   ```
   # Assume that GPR 5 contains the new value to be stored after
   # a successful match.
   # Assume that GPR 3 contains the address of the word
   # to be tested.
   # Assume that GPR 4 contains the value to be compared against
   # the value in memory.
   loop:   lwarx r6,0,r3  # Load and reserve
           cmpw r4,r6    # Are the first two operands
                       # equal?
           bne-  exit    # Skip if not equal
           stwcx. r5,0,r3 # Store new value if still
                       # reserved
           bne-  loop    # Loop if lost reservation
   exit:   mr r4,r6    # Return value from storage
       # The old value is returned to GPR 4.
       # If a match was made, storage contains the new value.
   ```

If the value in the register equals the word in storage, the value from a second register is stored in the word in storage. If they are unequal, the word from storage is loaded into the first register and the EQ bit of the Condition Register field 0 is set to indicate the result of the comparison.
Related Information
The **stwcx** (Store Word Conditional Indexed) instruction.

Processing and Storage

### Iwaux (Load Word Algebraic with Update Indexed) Instruction

**Purpose**
Load a fullword of data from storage into the low-order 32 bits of the specified general purpose register. Sign extend the data into the high-order 32 bits of the register. Update the address base.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-30</td>
<td>373</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

**Description**
The fullword in storage located at the effective address (EA) is loaded into the low-order 32 bits of the target general purpose register (GRP). The value is then sign-extended to fill the high-order 32 bits of the register. The EA is the sum of the contents of GRP RA and GRP RB.

If RA = 0 or RA = RT, the instruction form is invalid.

**Parameters**

- **RT** Specifies target general-purpose register where result of the operation is stored.
- **RA** Specifies source general-purpose register for EA calculation.
- **RB** Specifies source general-purpose register for EA calculation.

**Implementation**
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

### Iwax (Load Word Algebraic Indexed) Instruction

**Purpose**
Load a fullword of data from storage into the low-order 32 bits of the specified general purpose register. Sign extend the data into the high-order 32 bits of the register.
Syntac

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-30</td>
<td>341</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

POWER family
lwax  $RT$ $RA$ $RB$

Description
The fullword in storage located at the effective address (EA) is loaded into the low-order 32 bits of the target general purpose register (GRP). The value is then sign-extended to fill the high-order 32 bits of the register.

If GRP $RA$ is not 0, the EA is the sum of the contents of GRP $RA$ and $B$; otherwise, the EA is equal to the contents of $RB$.

Parameters
$RT$ Specifies target general-purpose register where result of operation is stored.
$RA$ Specifies source general-purpose register for EA calculation.
$RB$ Specifies source general-purpose register for EA calculation.

Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Iwbrx or Ibrx (Load Word Byte-Reversed Indexed) Instruction

Purpose
Loads a byte-reversed word of data from a specified location in memory into a general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>$RT$</td>
</tr>
<tr>
<td>11-15</td>
<td>$RA$</td>
</tr>
<tr>
<td>16-20</td>
<td>$RB$</td>
</tr>
<tr>
<td>21-30</td>
<td>534</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>
**Description**

The `lwbrx` and `lbrx` instructions load a byte-reversed word in storage from a specified location in memory addressed by the effective address (EA) into the target general-purpose register (GPR) `RT`.

Consider the following when using the `lwbrx` and `lbrx` instructions:

- Bits 00-07 of the word in storage addressed by EA are placed into bits 24-31 of GPR `RT`.
- Bits 08-15 of the word in storage addressed by EA are placed into bits 16-23 of GPR `RT`.
- Bits 16-23 of the word in storage addressed by EA are placed into bits 08-15 of GPR `RT`.
- Bits 24-31 of the word in storage addressed by EA are placed into bits 00-07 of GPR `RT`.

If GPR `RA` is not 0, the EA is the sum of the contents of GPR `RA` and GPR `RB`. If GPR `RA` is 0, then the EA is the contents of GPR `RB`.

The `lwbrx` and `lbrx` instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Parameters**

- `RT` Specifies target general-purpose register where result of operation is stored.
- `RA` Specifies source general-purpose register for EA calculation.
- `RB` Specifies source general-purpose register for EA calculation.

**Examples**

The following code loads a byte-reversed word from memory into GPR 6:

```assembly
storage: .long 0x0000 ffff.

# Assume GPR 4 contains 0x0000 0000.
# Assume GPR 5 contains address of storage.
lwbrx 6,4,5
# GPR 6 now contains 0xffff 0000.
```

**Related Information**

- [Fixed-Point Processor](#).
- [Fixed-Point Load and Store Instructions](#).

**Iwz or I (Load Word and Zero) Instruction**

**Purpose**

Loads a word of data from a specified location in memory into a general-purpose register.
### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>32</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

**PowerPC**

`lwz`  

`RT` `D` `RA`

**POWER family**

`I`  

`RT` `D` `RA`

### Description

The `lwz` and `I` instructions load a word in storage from a specified location in memory addressed by the effective address (EA) into the target general-purpose register (GPR) `RT`.

If GPR `RA` is not 0, the EA is the sum of the contents of GPR `RA` and `D`, a 16-bit, signed two's complement integer sign-extended to 32 bits. If GPR `RA` is 0, then the EA is `D`.

The `lwz` and `I` instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

### Parameters

- **RT** Specifies target general-purpose register where result of operation is stored.
- **D** Specifies a 16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
- **RA** Specifies source general-purpose register for EA calculation.

### Examples

The following code loads a word from memory into GPR 6:

```
.csect data[rw]
# Assume GPR 5 contains address of csect data[rw].
storage: .long 0x4
.csect text[pr]
lwz 6,storage(5)
# GPR 6 now contains 0x0000 0004.
```

### Related Information

- [Fixed-Point Processor](#).
- [Fixed-Point Load and Store Instructions](#).

### Iwzu or Iu (Load Word with Zero Update) Instruction

#### Purpose

Loads a word of data from a specified location in memory into a general-purpose register and possibly places the effective address in a second general-purpose register.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>33</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

PowerPC

\text{lwzu} \quad RT, D, RA

POWER family

\text{lu} \quad RT, D, RA

Description

The \text{lwzu} and \text{lu} instructions load a word in storage from a specified location in memory addressed by the effective address (EA) into the target general-purpose register (GPR) \text{RT}.

If GPR \text{RA} is not 0, the EA is the sum of the contents of GPR \text{RA} and \text{D}, a 16-bit, signed two’s complement integer sign-extended to 32 bits. If GPR \text{RA} is 0, then the EA is \text{D}.

If \text{RA} does not equal \text{RT} and \text{RA} does not equal 0, and the storage access does not cause an Alignment interrupt or a Data Storage interrupt, then the EA is placed into GPR \text{RA}.

The \text{lwzu} and \text{lu} instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

\text{RT} \quad \text{Specifies target general-purpose register where result of operation is stored.}

\text{D} \quad \text{Specifies a 16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.}

\text{RA} \quad \text{Specifies source general-purpose register for EA calculation and possible address update.}

Examples

The following code loads a word from memory into GPR 6 and places the effective address in GPR 4:

```
.csect data[rw]
storage: .long 0xffdd 75ce
.csect text[pr]
# Assume GPR 4 contains address of csect data[rw].
lwzu 6, storage(4)
# GPR 6 now contains 0xffdd 75ce.
# GPR 4 now contains the storage address.
```

Related Information

[Fixed-Point Processor]

[Fixed-Point Load and Store with Update Instructions]
Iwzux or lux (Load Word and Zero with Update Indexed) Instruction

Purpose
Loads a word of data from a specified location in memory into a general-purpose register and possibly places the effective address in a second general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>55</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

PowerPC
lwzux \( RT, RA, RB \)

POWER family
lux \( RT, RA, RB \)

Description
The lwzux and lux instructions load a word of data from a specified location in memory, addressed by the effective address (EA), into the target general-purpose register (GPR) \( RT \).

If GPR \( RA \) is not 0, the EA is the sum of the contents of GPR \( RA \) and GPR \( RB \). If GPR \( RA \) is 0, then the EA is the contents of GPR \( RB \).

If GPR \( RA \) does not equal \( RT \) and \( RA \) does not equal 0, and the storage access does not cause an Alignment interrupt or a Data Storage interrupt, then the EA is placed into GPR \( RA \).

The lwzux and lux instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters
\( RT \) Specifies target general-purpose register where result of operation is stored.
\( RA \) Specifies source general-purpose register for EA calculation and possible address update.
\( RB \) Specifies source general-purpose register for EA calculation.

Examples
The following code loads a word from memory into GPR 6 and places the effective address in GPR 5:

```assembly
.csect data[rw]
storage: .long 0xffdd 75ce
# Assume GPR 5 contains the address of csect data[rw].
# Assume GPR 4 contains the displacement of storage
# relative to csect data[rw].
```
Related Information

Fixed-Point Processor.

Fixed-Point Load and Store with Update Instructions.

Iwzx or Ix (Load Word and Zero Indexed) Instruction

Purpose
Loads a word of data from a specified location in memory into a general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>23</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

PowerPC

Iwzx

POWER family

Ix

Description
The iwzx and Ix instructions load a word of data from a specified location in memory, addressed by the effective address (EA), into the target general-purpose register (GPR) RT.

If GPR RA is not 0, the EA is the sum of the contents of GPR RA and GPR RB. If GPR RA is 0, then the EA is the contents of GPR RB.

The iwzx and Ix instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

RT Specifies target general-purpose register where result of operation is stored.
RA Specifies source general-purpose register for EA calculation.
RB Specifies source general-purpose register for EA calculation.

Examples
The following code loads a word from memory into GPR 6:
Related Information

Fixed-Point Processor

Fixed-Point Load and Store Instructions

maskg (Mask Generate) Instruction

Purpose
Generates a mask of ones and zeros and loads it into a general-purpose register.

Note: The maskg instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>29</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family

maskg

Description
The maskg instruction generates a mask from a starting point defined by bits 27-31 of general-purpose register (GPR) RS to an end point defined by bits 27-31 of GPR RB and stores the mask in GPR RA.

Consider the following when using the maskg instruction:
- If the starting point bit is less than the end point bit + 1, then the bits between and including the starting point and the end point are set to ones. All other bits are set to 0.
- If the starting point bit is the same as the end point bit + 1, then all 32 bits are set to ones.
- If the starting point bit is greater than the end point bit + 1, then all of the bits between and including the end point bit + 1 and the starting point bit - 1 are set to zeros. All other bits are set to ones.

The maskg instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.
The two syntax forms of the `maskg` instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

### Parameters

- **RA**: Specifies target general-purpose register where result of operation is stored.
- **RS**: Specifies source general-purpose register for start of mask.
- **RB**: Specifies source general-purpose register for end of mask.

### Examples

1. The following code generates a mask of 5 ones and stores the result in GPR 6:
   
   ```
   # Assume GPR 4 contains 0x0000 0014.
   # Assume GPR 5 contains 0x0000 0010.
   maskg 6,5,4
   # GPR 6 now contains 0x0000 F800.
   ```

2. The following code generates a mask of 6 zeros with the remaining bits set to one, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x0000 0010.
   # Assume GPR 5 contains 0x0000 0017.
   # Assume CR = 0.
   maskg 6,5,4
   # GPR 6 now contains 0xFFFF 81FF.
   # CR now contains 0x8000 0000.
   ```

### Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Rotate and Shift Instructions](#)

---

**maskir** *(Mask Insert from Register) Instruction*

### Purpose

Inserts the contents of one general-purpose register into another general-purpose register under control of a bit mask.

- **Note**: The `maskir` instruction is supported only in the POWER family architecture.

### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>541</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
Description

The **maskir** stores the contents of general-purpose register (GPR) **RS** in GPR **RA** under control of the bit mask in GPR **RB**.

The value for each bit in the target GPR **RA** is determined as follows:
- If the corresponding bit in the mask GPR **RB** is 1, then the bit in the target GPR **RA** is given the value of the corresponding bit in the source GPR **RS**.
- If the corresponding bit in the mask GPR **RB** is 0, then the bit in the target GPR **RA** is unchanged.

The **maskir** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>maskir</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>maskir.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT, GT, EQ, SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the **maskir** instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

- **RA** Specifies target general-purpose register where result of operation is stored.
- **RS** Specifies source general-purpose register for operation.
- **RB** Specifies source general-purpose register for bit mask.

Examples

1. The following code inserts the contents of GPR 5 into GPR 6 under control of the bit mask in GPR 4:

   ```
   # Assume GPR 6 (RA) target contains 0xAAAAAAAA.
   # Assume GPR 4 (RB) mask contains 0x000F0F00.
   # Assume GPR 5 (RS) source contains 0x55555555.
   maskir 6,5,4
   # GPR 6 (RA) target now contains 0xAAA5A5AA.
   ```

1. The following code inserts the contents of GPR 5 into GPR 6 under control of the bit mask in GPR 4 and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 6 (RA) target contains 0xAAAAAAAA.
   # Assume GPR 4 (RB) mask contains 0x0A050F00.
   # Assume GPR 5 (RS) source contains 0x55555555.
   maskir. 6,5,4
   # GPR 6 (RA) target now contains 0xA0AFA5AA.
   ```

Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Rotate and Shift Instructions](#)
mcrf (Move Condition Register Field) Instruction

Purpose
Copies the contents of one condition register field into another.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>19</td>
</tr>
<tr>
<td>6-8</td>
<td>BF</td>
</tr>
<tr>
<td>9-10</td>
<td>//</td>
</tr>
<tr>
<td>11-13</td>
<td>BFA</td>
</tr>
<tr>
<td>14-15</td>
<td>//</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

mcrf BF, BFA

Description
The mcrf instruction copies the contents of the condition register field specified by BFA into the condition register field specified by BF. All other fields remain unaffected.

The mcrf instruction has one syntax form and does not affect Condition Register Field 0 or the Fixed-Point Exception Register.

Parameters

BF Specifies target condition register field for operation.
BFA Specifies source condition register field for operation.

Examples
The following code copies the contents of Condition Register Field 3 into Condition Register Field 2:

```plaintext
# Assume Condition Register Field 3 holds b'0110'.
mcrf 2,3
# Condition Register Field 2 now holds b'0110'.
```

Related Information

Branch Processor

mcrfs (Move to Condition Register from FPSCR) Instruction

Purpose
Copies the bits from one field of the Floating-Point Status and Control Register into the Condition Register.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-8</td>
<td>BF</td>
</tr>
<tr>
<td>9-10</td>
<td>//</td>
</tr>
<tr>
<td>11-13</td>
<td>BFA</td>
</tr>
<tr>
<td>14-15</td>
<td>//</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>64</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

mcrfs

Description

The `mcrfs` instruction copies four bits of the Floating-Point Status and Control Register (FPSCR) specified by `BFA` into Condition Register Field `BF`. All other Condition Register bits are unchanged.

If the field specified by `BFA` contains reserved or undefined bits, then bits of zero value are supplied for the copy.

The `mcrfs` instruction has one syntax form and can set the bits of the Floating-Point Status and Control Register.

<table>
<thead>
<tr>
<th>BFA</th>
<th>FPSCR bits set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FX,OX</td>
</tr>
<tr>
<td>1</td>
<td>UX, ZX, XX, VXSNAN</td>
</tr>
<tr>
<td>2</td>
<td>VXISI, VXIDI, VXZDZ, VXIMZ</td>
</tr>
<tr>
<td>3</td>
<td>VXVC</td>
</tr>
</tbody>
</table>

Parameters

- `BF` Specifies target condition register field where result of operation is stored.
- `BFA` Specifies one of the FPSCR fields (0-7).

Examples

The following code copies bits from Floating-Point Status and Control Register Field 4 into Condition Register Field 3:

```c
# Assume FPSCR 4 contains b'0111'.
mcrfs 3,4
# Condition Register Field 3 contains b'0111'.
```

Related Information

- [Branch Processor](#)
- [Interpreting the Contents of a Floating-Point Register](#)
**mcrxr (Move to Condition Register from XER) Instruction**

**Purpose**
Copies the Summary Overflow bit, Overflow bit, Carry bit, and bit 3 from the Fixed-Point Exception Register into a specified field of the Condition Register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-8</td>
<td>BF</td>
</tr>
<tr>
<td>9-10</td>
<td>//</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>512</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

**mcrxr BF**

**Description**
The `mcrxr` copies the contents of Fixed-Point Exception Register Field 0 bits 0-3 into Condition Register Field `BF` and resets Fixed-Point Exception Register Field 0 to 0.

The `mcrxr` instruction has one syntax form and resets Fixed-Point Exception Register bits 0-3 to 0.

**Parameters**

`BF` Specifies target condition register field where result of operation is stored.

**Examples**
The following code copies the Summary Overflow bit, Overflow bit, Carry bit, and bit 3 from the Fixed-Point Exception Register into field 4 of the Condition Register.

```assembly
# Assume bits 0-3 of the Fixed-Point Exception
# Register are set to b'1110'.
mcrxr 4
# Condition Register Field 4 now holds b'1110'.
```

**Related Information**

- [Branch Processor](#)

  [Fixed-Point Move to or from Special-Purpose Registers Instructions](#)

**mfcfr (Move from Condition Register) Instruction**

**Purpose**
Copies the contents of the Condition Register into a general-purpose register.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>19</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

mfcr  

Description
The **mfcr** instruction copies the contents of the Condition Register into target general-purpose register (GPR) *RT*.

The **mfcr** instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters

*RT*  
Specifies target general-purpose register where result of operation is stored.

Examples

The following code copies the Condition Register into GPR 6:

```
# Assume the Condition Register contains 0x4055 F605.
mfcr 6
# GPR 6 now contains 0x4055 F605.
```

Related Information

- [Branch Processor](#)
- [Fixed-Point Move to or from Special-Purpose Registers Instructions](#)

mffs (Move from FPSCR) Instruction

Purpose

Loads the contents of the Floating-Point Status and Control Register into a floating-point register and fills the upper 32 bits with ones.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>FRT</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>583</td>
</tr>
</tbody>
</table>
### Description

The mffs instruction places the contents of the Floating-Point Status and Control Register into bits 32-63 of floating-point register (FPR) $FRT$. The bits 0-31 of floating-point register $FRT$ are undefined.

The mffs instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>FPSCR bits</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mffs</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>mffs.</td>
<td>None</td>
<td>1</td>
<td>FX, FEX, VX, OX</td>
</tr>
</tbody>
</table>

The two syntax forms of the mffs instruction never affect the Floating-Point Status and Control Register fields. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

### Parameters

$FRT$  
Specifies target floating-point register where result of operation is stored.

### Examples

The following code loads the contents of the Floating-Point Status and Control Register into FPR 14, and fills the upper 32 bits of that register with ones:

```plaintext
# Assume FPSCR contains 0x0000 0000.
mffs 14
# FPR 14 now contains 0xFFFF FFFF 0000 0000.
```

### Related Information

- [Floating-Point Processor](#)
- [Interpreting the Contents of a Floating-Point Register](#)
- [Functional Differences for POWER family and PowerPC Instructions](#)

### mfmsr (Move from Machine State Register) Instruction

**Purpose**

Copies the contents of the Machine State Register into a general-purpose register.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>83</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

`mfmsr`  

Description

The `mfmsr` instruction copies the contents of the Machine State Register into the target general-purpose register (GPR) `RT`.

The `mfmsr` instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

`RT` Specifies target general-purpose register where result of operation is stored.

Examples

The following code copies the contents of the Machine State Register into GPR 4:

```
mfmsr 4
```

# GPR 4 now holds a copy of the bit
# settings of the Machine State Register.

Security

The `mfmsr` instruction is privileged only in the PowerPC architecture.

Related Information

- [Branch Processor](#)
- [Floating-Point Processor](#)
- [Fixed-Point Move to or from Special-Purpose Registers Instructions](#)
- [Functional Differences for POWER family and PowerPC Instructions](#)

`mfocrf` (Move from One Condition Register Field) Instruction

Purpose

Copies the contents of one Condition Register field into a general-purpose register.
### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>12-19</td>
<td>FXM</td>
</tr>
<tr>
<td>20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>19</td>
</tr>
<tr>
<td>31</td>
<td>///</td>
</tr>
</tbody>
</table>

**mfocrf**

#### Description

The **mfocrf** instruction copies the contents of one Condition Register field specified by the field mask FXM into the target general-purpose register (GPR) **RT**.

Field mask FXM is defined as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>CR 00-03 is copied into GPR RS 00-03.</td>
</tr>
<tr>
<td>13</td>
<td>CR 04-07 is copied into GPR RS 04-07.</td>
</tr>
<tr>
<td>14</td>
<td>CR 08-11 is copied into GPR RS 08-11.</td>
</tr>
<tr>
<td>15</td>
<td>CR 12-15 is copied into GPR RS 12-15.</td>
</tr>
<tr>
<td>16</td>
<td>CR 16-19 is copied into GPR RS 16-19.</td>
</tr>
<tr>
<td>17</td>
<td>CR 20-23 is copied into GPR RS 20-23.</td>
</tr>
<tr>
<td>18</td>
<td>CR 24-27 is copied into GPR RS 24-27.</td>
</tr>
<tr>
<td>19</td>
<td>CR 28-31 is copied into GPR RS 28-31.</td>
</tr>
</tbody>
</table>

The **mfocrf** instruction has one syntax form and does not affect the Fixed-Point Exception Register.

#### Parameters

**RT**  
Specifies target general-purpose register where result of operation is stored.

**FXM**  
Specifies field mask. Only one bit may be specified.

#### Examples

The following code copies the Condition Register field 3 into GPR 6:

```assembly
# Assume the Condition Register contains 0x4055 F605.
# Field 3 (0x10 = b'0001 0000')
mfocrf 6, 0x10
# GPR 6 now contains 0x0005 0000.
```

#### Related Information

- "Branch Processor" on page 19.
- "Fixed-Point Move to or from Special-Purpose Registers Instructions" on page 23.
mfspr (Move from Special-Purpose Register) Instruction

Purpose
Copies the contents of a special-purpose register into a general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-20</td>
<td>spr</td>
</tr>
<tr>
<td>21-30</td>
<td>339</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

mfspr  

**Note:** The special-purpose register is a split field.

See [Extended Mnemonics of Moving from or to Special-Purpose Registers](#) for more information.

Description
The mfspr instruction copies the contents of the special-purpose register SPR into target general-purpose register (GPR) RT.

The special-purpose register identifier SPR can have any of the values specified in the following table. The order of the two 5-bit halves of the SPR number is reversed.

<table>
<thead>
<tr>
<th>SPR Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decimal</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>26</td>
</tr>
<tr>
<td>27</td>
</tr>
<tr>
<td>272</td>
</tr>
<tr>
<td>273</td>
</tr>
<tr>
<td>274</td>
</tr>
<tr>
<td>275</td>
</tr>
<tr>
<td>282</td>
</tr>
<tr>
<td>284</td>
</tr>
</tbody>
</table>
The following code copies the contents of the Fixed-Point Exception Register into GPR 6:

```assembly
mfspr 6,1
```

*No* GPR 6 now contains the bit settings of the Fixed Point Exception Register.
Related Information

**Fixed-Point Processor**.

**Fixed-Point Move to or from Special-Purpose Registers Instructions**.

---

**mfsr (Move from Segment Register) Instruction**

**Purpose**

Copies the contents of a segment register into a general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-8</td>
<td>RT</td>
</tr>
<tr>
<td>11</td>
<td>/</td>
</tr>
<tr>
<td>12-14</td>
<td>SR</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>595</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

\[
mfsr \ RT, \ SR
\]

**Description**

The `mfsr` instruction copies the contents of segment register (SR) into target general-purpose register (GPR) `RT`.

The `mfsr` instruction has one syntax form and does not effect the Fixed-Point Exception Register. If the Record (Rc) bit is set to 1, Condition Register Field 0 is undefined.

**Parameters**

- `RT` Specifies the target general-purpose register where the result of the operation is stored.
- `SR` Specifies the source segment register for the operation.

**Examples**

The following code copies the contents of Segment Register 7 into GPR 6:

```plaintext
# Assume that the source Segment Register is SR 7.
# Assume that GPR 6 is the target register.
mfsr 6,7
# GPR 6 now holds a copy of the contents of Segment Register 7.
```

**Security**

The `mfsr` instruction is privileged only in the PowerPC architecture.

**Related Information**

The `mfsri` (Move from Segment Register Indirect) instruction, `mtsr` (Move to Segment Register) instruction, `mtsri` or `mtsri` (Move to Segment Register Indirect) instruction.
mfsri (Move from Segment Register Indirect) Instruction

Purpose
Copies the contents of a calculated segment register into a general-purpose register.

Note: The mfsri instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>627</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family
mfsri  RS, RA, RB

Description
The mfsri instruction copies the contents of segment register (SR), specified by bits 0-3 of the calculated contents of the general-purpose register (GPR) RA, into GPR RS. If RA is not 0, the specifying bits in GPR RA are calculated by adding the original contents of RA to GPR RB and placing the sum in RA. If RA = RS, the sum is not placed in RA.

The mfsri instruction has one syntax form and does not affect the Fixed-Point Exception Register. If the Record (Rc) bit is set to 1, Condition Register Field 0 is undefined.

Parameters

RS  Specifies the target general-purpose register for operation.
RA  Specifies the source general-purpose register for SR calculation.
RB  Specifies the source general-purpose register for SR calculation.

Examples
The following code copies the contents of the segment register specified by the first 4 bits of the sum of the contents of GPR 4 and GPR 5 into GPR 6:

# Assume that GPR 4 contains 0x9000 3000.
# Assume that GPR 5 contains 0x1000 0000.
# Assume that GPR 6 is the target register.
mfsri 6,5,4
# GPR 6 now contains the contents of Segment Register 10.
Related Information

The **mfsrin** (Move from Segment Register Indirect) instruction, **mtsr** (Move to Segment Register) instruction, **mfsri** or **mtsri** (Move to Segment Register Indirect) instruction.

**mfsrin (Move from Segment Register Indirect) Instruction**

**Purpose**
Copies the contents of the specified segment register into a general-purpose register.

*Note:* The **mfsrin** instruction is supported only in the PowerPC architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>659</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

**Description**
The **mfsrin** instruction copies the contents of segment register (SR), specified by bits 0-3 of the general-purpose register (GPR) *RB*, into GPR *RT*.

The **mfsrin** instruction has one syntax form and does not affect the Fixed-Point Exception Register. If the Record (Rc) bit is set to 1, the Condition Register Field 0 is undefined.

**Parameters**

*RT*  Specifies the target general-purpose register for operation.

*RB*  Specifies the source general-purpose register for SR calculation.

**Security**
The **mfsrin** instruction is privileged.

**Related Information**
The **mfsr** (Move from Segment Register) instruction, **mfsri** (Move from Segment Register Indirect) instruction, **mtsr** (Move to Segment Register) instruction, **mfsri** or **mtsri** (Move to Segment Register Indirect) instruction.
**mtcrf (Move to Condition Register Fields) Instruction**

**Purpose**
Copies the contents of a general-purpose register into the condition register under control of a field mask.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11</td>
<td>/</td>
</tr>
<tr>
<td>12-19</td>
<td>FXM</td>
</tr>
<tr>
<td>20</td>
<td>/</td>
</tr>
<tr>
<td>21-30</td>
<td>144</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

```
mtcrf [FXM]RS
```

See [Extended Mnemonics of Condition Register Logical Instructions](#) for more information.

**Description**
The `mtcrf` instruction copies the contents of source general-purpose register (GPR) `RS` into the condition register under the control of field mask `FXM`.

Field mask `FXM` is defined as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>CR 00-03 is updated with the contents of GPR <code>RS</code> 00-03.</td>
</tr>
<tr>
<td>13</td>
<td>CR 04-07 is updated with the contents of GPR <code>RS</code> 04-07.</td>
</tr>
<tr>
<td>14</td>
<td>CR 08-11 is updated with the contents of GPR <code>RS</code> 08-11.</td>
</tr>
<tr>
<td>15</td>
<td>CR 12-15 is updated with the contents of GPR <code>RS</code> 12-15.</td>
</tr>
<tr>
<td>16</td>
<td>CR 16-19 is updated with the contents of GPR <code>RS</code> 16-19.</td>
</tr>
<tr>
<td>17</td>
<td>CR 20-23 is updated with the contents of GPR <code>RS</code> 20-23.</td>
</tr>
<tr>
<td>18</td>
<td>CR 24-27 is updated with the contents of GPR <code>RS</code> 24-27.</td>
</tr>
<tr>
<td>19</td>
<td>CR 28-31 is updated with the contents of GPR <code>RS</code> 28-31.</td>
</tr>
</tbody>
</table>

The `mtcrf` instruction has one syntax form and does not affect the Fixed-Point Exception Register.

The preferred form of the `mtcrf` instruction has only one bit set in the `FXM` field.

**Parameters**

- `FXM` Specifies field mask.
- `RS` Specifies source general-purpose register for operation.

**Examples**
The following code copies bits 00-03 of GPR 5 into Condition Register Field 0:
# Assume GPR 5 contains 0x7542 FFEE.
# Use the mask for Condition Register
# Field 0 (0x80 = b'1000 0000').
mctrf 0x80,5
# Condition Register Field 0 now contains b'0111'.

## Related Information

- [Fixed-Point Processor](#)
- [Branch Processor](#)
- [Fixed-Point Move to or from Special-Purpose Registers Instructions](#)

### mtfsb0 (Move to FPSCR Bit 0) Instruction

#### Purpose
Sets a specified Floating-Point Status and Control Register bit to 0.

#### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>BT</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>70</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

- mtfsb0
- mtfsb0.

#### Description

The *mtfsb0* instruction sets the Floating-Point Status and Control Register bit specified by *BT* to 0.

The *mtfsb0* instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtfsb0</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>mtfsb0.</td>
<td>None</td>
<td>1</td>
<td>FX, FEX, VX, OX</td>
</tr>
</tbody>
</table>

The two syntax forms of the *mtfsb0* instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

**Note:** Bits 1-2 cannot be explicitly set or reset.
Parameters

$BT$  Specifies Floating-Point Status and Control Register bit set by operation.

Examples

1. The following code sets the Floating-Point Status and Control Register Floating-Point Overflow Exception Bit (bit 3) to 0:

   ```
   mtfsb0 3
   # Now bit 3 of the Floating-Point Status and Control
   # Register is 0.
   ```

2. The following code sets the Floating-Point Status and Control Register Floating-Point Overflow Exception Bit (bit 3) to 0 and sets Condition Register Field 1 to reflect the result of the operation:

   ```
   mtfsb0. 3
   # Now bit 3 of the Floating-Point Status and Control
   # Register is 0.
   ```

Related Information

Floating-Point Processor.
Interpreting the Contents of a Floating-Point Register.

mtfsb1 (Move to FPSCR Bit 1) Instruction

Purpose

Sets a specified Floating-Point Status and Control Register bit to 1.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-10</td>
<td>$BT$</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>38</td>
</tr>
<tr>
<td>31</td>
<td>$Rc$</td>
</tr>
</tbody>
</table>

mtfsb1  
mtfsb1.  

Description

The `mtfsb1` instruction sets the Floating-Point Status and Control Register (FPSCR) bit specified by $BT$ to 1.

The `mtfsb1` instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>FPSCR Bits</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtfsb1</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>
The two syntax forms of the `mtfsb1` instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

**Note:** Bits 1-2 cannot be explicitly set or reset.

### Parameters

**BT** Specifies that the FPSCR bit is set to 1 by instruction.

### Examples

1. The following code sets the Floating-Point Status and Control Register bit 4 to 1:
   ```
   mtfsb1 4
   # Now bit 4 of the Floating-Point Status and Control
   # Register is set to 1.
   ```
2. The following code sets the Floating-Point Status and Control Register Overflow Exception Bit (bit 3) to 1 and sets Condition Register Field 1 to reflect the result of the operation:
   ```
   mtfsb1. 3
   # Now bit 3 of the Floating-Point Status and Control
   # Register is set to 1.
   ```

### Related Information

- [Floating-Point Processor](#)
- [Interpreting the Contents of a Floating-Point Register](#)

---

## mtfsf (Move to FPSCR Fields) Instruction

### Purpose

Copies the contents of a floating-point register into the Floating-Point Status and Control Register under the control of a field mask.

### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7-14</td>
<td>FLM</td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16-20</td>
<td>FRB</td>
</tr>
<tr>
<td>21-30</td>
<td>771</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

```

mtfsf  
mtfsf.
```
See [Extended Mnemonics of Condition Register Logical Instructions](#) for more information.

**Description**

The `mtfsf` instruction copies bits 32-63 of the contents of the floating-point register (FPR) `FRB` into the Floating-Point Status and Control Register under the control of the field mask specified by `FLM`.

The field mask `FLM` is defined as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>FPSCR 00-03 is updated with the contents of <code>FRB</code> 32-35.</td>
</tr>
<tr>
<td>8</td>
<td>FPSCR 04-07 is updated with the contents of <code>FRB</code> 36-39.</td>
</tr>
<tr>
<td>9</td>
<td>FPSCR 08-11 is updated with the contents of <code>FRB</code> 40-43.</td>
</tr>
<tr>
<td>10</td>
<td>FPSCR 12-15 is updated with the contents of <code>FRB</code> 44-47.</td>
</tr>
<tr>
<td>11</td>
<td>FPSCR 16-19 is updated with the contents of <code>FRB</code> 48-51.</td>
</tr>
<tr>
<td>12</td>
<td>FPSCR 20-23 is updated with the contents of <code>FRB</code> 52-55.</td>
</tr>
<tr>
<td>13</td>
<td>FPSCR 24-27 is updated with the contents of <code>FRB</code> 56-59.</td>
</tr>
<tr>
<td>14</td>
<td>FPSCR 28-31 is updated with the contents of <code>FRB</code> 60-63.</td>
</tr>
</tbody>
</table>

The `mtfsf` instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>FPSCR Bits</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mtfsf</code></td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>mtfsf.</code></td>
<td>None</td>
<td>1</td>
<td>FX, FEX, VX, OX</td>
</tr>
</tbody>
</table>

The two syntax forms of the `mtfsf` instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.

**Note:** When specifying FPSCR 0-3, some bits cannot be explicitly set or reset.

**Parameters**

- `FLM` Specifies field mask.
- `FRB` Specifies source floating-point register for operation.

**Examples**

1. The following code copies the contents of floating-point register 5 bits 32-35 into Floating-Point Status and Control Register Field 0:
   ```assembly
   # Assume bits 32-63 of FPR 5
   # contain 0x3000 3000.
   mtfsf 0x80,5
   # Floating-Point Status and Control Register
   # Field 0 is set to b'0001'.
   ```

2. The following code copies the contents of floating-point register 5 bits 32-43 into Floating-Point Status and Control Register Fields 0-2 and sets Condition Register Field 1 to reflect the result of the operation:
Related Information

Floating-Point Processor

Interpreting the Contents of a Floating-Point Register

**mtfsfi (Move to FPSCR Field Immediate) Instruction**

**Purpose**

Copies an immediate value into a specified Floating-Point Status and Control Register field.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>63</td>
</tr>
<tr>
<td>6-8</td>
<td>BF</td>
</tr>
<tr>
<td>9-10</td>
<td>//</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-19</td>
<td>U</td>
</tr>
<tr>
<td>20</td>
<td>/</td>
</tr>
<tr>
<td>21-30</td>
<td>134</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**mtfsfi**

**mtfsfi.**

**Description**

The **mtfsfi** instruction copies the immediate value specified by the \( I \) parameter into the Floating-Point Status and Control Register field specified by \( BF \). None of the other fields of the Floating-Point Status and Control Register are affected.

The **mtfsfi** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>FPSCR Bits</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtfsfi</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>mtfsfi.</td>
<td>None</td>
<td>1</td>
<td>FX, FEX, VX, OX</td>
</tr>
</tbody>
</table>

The two syntax forms of the **mtfsfi** instruction never affect the Floating-Point Status and Control Register fields. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating-Point Exception (FX), Floating-Point Enabled Exception (FEX), Floating Invalid Operation Exception (VX), and Floating-Point Overflow Exception (OX) bits in Condition Register Field 1.
Note: When specifying FPSCR 0-3, some bits cannot be explicitly set or reset.

Parameters

\[ BF \] Specifies target Floating-Point Status and Control Register field for operation.
\[ I \] Specifies source immediate value for operation.

Examples

1. The following code sets Floating-Point Status and Control Register Field 6 to b'0100':
   ```
   mtfsfi 6,4
   # Floating-Point Status and Control Register Field 6
   # is now b'0100'.
   ```

2. The following code sets Floating-Point Status and Control Register field 0 to b'0100' and sets Condition Register Field 1 to reflect the result of the operation:
   ```
   mtfsfi. 0,1
   # Floating-Point Status and Control Register Field 0
   # is now b'0001'.
   # Condition Register Field 1 now contains 0x1.
   ```

Related Information

- Floating-Point Processor
- Interpreting the Contents of a Floating-Point Register

\textbf{mtocrf (Move to One Condition Register Field) Instruction}

**Purpose**
Copies the contents of a general-purpose register into one condition register field under control of a field mask.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11</td>
<td>/</td>
</tr>
<tr>
<td>12-19</td>
<td>FXM</td>
</tr>
<tr>
<td>20</td>
<td>/</td>
</tr>
<tr>
<td>21-30</td>
<td>144</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

mtocrf \textit{"FXM" on page 315, "RS" on page 315}

See \textit{"Extended Mnemonics of Condition Register Logical Instructions" on page 96} for more information.

**Description**
The \textit{mtocrf} instruction copies the contents of source general-purpose register (GPR) \textit{RS} into the condition register under the control of field mask \textit{FXM}.  

314 Assembler Language Reference
Field mask *FXM* is defined as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>CR 00-03 is updated with the contents of GPR RS 00-03.</td>
</tr>
<tr>
<td>13</td>
<td>CR 04-07 is updated with the contents of GPR RS 04-07.</td>
</tr>
<tr>
<td>14</td>
<td>CR 08-11 is updated with the contents of GPR RS 08-11.</td>
</tr>
<tr>
<td>15</td>
<td>CR 12-15 is updated with the contents of GPR RS 12-15.</td>
</tr>
<tr>
<td>16</td>
<td>CR 16-19 is updated with the contents of GPR RS 16-19.</td>
</tr>
<tr>
<td>17</td>
<td>CR 20-23 is updated with the contents of GPR RS 20-23.</td>
</tr>
<tr>
<td>18</td>
<td>CR 24-27 is updated with the contents of GPR RS 24-27.</td>
</tr>
<tr>
<td>19</td>
<td>CR 28-31 is updated with the contents of GPR RS 28-31.</td>
</tr>
</tbody>
</table>

The **mtocrf** instruction has one syntax form and does not affect the Fixed-Point Exception Register.

**Parameters**

- *FXM*  
  Specifies field mask.
- *RS*  
  Specifies source general-purpose register for operation.

**Examples**

The following code copies bits 00-03 of GPR 5 into Condition Register Field 0:

```
# Assume GPR 5 contains 0x7542 FFEE.
# Use the mask for Condition Register
# Field 0 (0x80 = b'1000 0000').
mtocrf 0x80,5
# Condition Register Field 0 now contains b'0111'.
```

**Related Information**

- [“Fixed-Point Processor” on page 21](#)
- [“Branch Processor” on page 19](#)
- [“Fixed-Point Move to or from Special-Purpose Registers Instructions” on page 23](#)

**mtspr (Move to Special-Purpose Register) Instruction**

**Purpose**

Copies the contents of a general-purpose register into a special-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-20</td>
<td>spr</td>
</tr>
<tr>
<td>21-30</td>
<td>467</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
Note: The special-purpose register is a split field.

See [Extended Mnemonics of Moving from or to Special-Purpose Registers](#) for more information.

### Description

The `mtspr` instruction copies the contents of the source general-purpose register `RS` into the target special-purpose register `SPR`.

The special-purpose register identifier `SPR` can have any of the values specified in the following table. The order of the two 5-bit halves of the SPR number is reversed.

<table>
<thead>
<tr>
<th>SPR Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>26</td>
</tr>
<tr>
<td>27</td>
</tr>
<tr>
<td>272</td>
</tr>
<tr>
<td>274</td>
</tr>
<tr>
<td>275</td>
</tr>
<tr>
<td>282</td>
</tr>
<tr>
<td>284</td>
</tr>
<tr>
<td>285</td>
</tr>
<tr>
<td>528</td>
</tr>
<tr>
<td>529</td>
</tr>
<tr>
<td>530</td>
</tr>
<tr>
<td>531</td>
</tr>
<tr>
<td>532</td>
</tr>
<tr>
<td>533</td>
</tr>
<tr>
<td>534</td>
</tr>
<tr>
<td>535</td>
</tr>
<tr>
<td>536</td>
</tr>
<tr>
<td>537</td>
</tr>
<tr>
<td>538</td>
</tr>
<tr>
<td>539</td>
</tr>
</tbody>
</table>
1. Moving to the DEC register is privileged in the PowerPC architecture and in the POWER family architecture. However, moving from the DEC register is privileged only in the PowerPC architecture.

2. Supported only in the POWER family architecture.

If the SPR field contains any value other than those listed in the SPR Values table, the instruction form is invalid.

The `mtspr` instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Parameters**

- **SPR**: Specifies target special-purpose register for operation.
- **RS**: Specifies source general-purpose register for operation.

**Examples**

The following code copies the contents of GPR 5 into the Link Register:

```plaintext
mtspr 8,5
```

Note: The `mtspr` instruction is supported only in the POWER family architecture.

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Move to or from Special-Purpose Registers Instructions](#)

**mul (Multiply) Instruction**

**Purpose**

Multiplies the contents of two general-purpose registers and stores the result in a third general-purpose register.

**Note**: The `mul` instruction is supported only in the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
</tbody>
</table>
### Description

The **mul** instruction multiplies the contents of general-purpose register (GPR) *RA* and GPR *RB*, and stores bits 0-31 of the result in the target GPR *RT* and bits 32-63 of the result in the MQ Register.

The **mul** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mul</strong></td>
<td>0</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><strong>mul.</strong></td>
<td>0</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td><strong>mulo</strong></td>
<td>1</td>
<td>SO,OV</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><strong>mulo.</strong></td>
<td>1</td>
<td>SO,OV</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The four syntax forms of the **mul** instruction never affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction sets the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register to 1 if the product is greater than 32 bits. If the syntax form sets the Record (Rc) bit to 1, then the Less Than (LT) zero, Greater Than (GT) zero and Equal To (EQ) zero bits in Condition Register Field 0 reflect the result in the low-order 32 bits of the MQ Register.

### Parameters

- **RT** Specifies target general-purpose register where the result of operation is stored.
- **RA** Specifies source general-purpose register for operation.
- **RB** Specifies source general-purpose register for operation.

### Examples

1. The following code multiplies the contents of GPR 4 by the contents of GPR 10 and stores the result in GPR 6 and the MQ Register:
   ```assembly
   # Assume GPR 4 contains 0x0000 0003.
   # Assume GPR 10 contains 0x0000 0002.
   mul 6,4,10
   # MQ Register now contains 0x0000 0006.
   # GPR 6 now contains 0x0000 0000.
   ```

2. The following code multiplies the contents of GPR 4 by the contents of GPR 10, stores the result in GPR 6 and the MQ Register, and sets Condition Register Field 0 to reflect the result of the operation:
3. The following code multiplies the contents of GPR 4 by the contents of GPR 10, stores the result in GPR 6 and the MQ Register, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x0000 4500.
# Assume GPR 10 contains 0x8000 7000.
mul. 6,4,10
# MQ Register now contains 0x1E30 0000.
# GPR 6 now contains 0xFFFF DD80.
# Condition Register Field 0 now contains 0x4.
```

4. The following code multiplies the contents of GPR 4 by the contents of GPR 10, stores the result in GPR 6 and the MQ Register, and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x0000 4500.
# Assume GPR 10 contains 0x8000 7000.
# Assume XER = 0.
mulo 6,4,10
# MQ Register now contains 0x1E30 0000.
# GPR 6 now contains 0xFFFF DD80.
# XER now contains 0xc000 0000.
# Condition Register Field 0 now contains 0x5.
```

### Related Information

The `mulhw` (Multiply High Word) instruction, `mulhwu` (Multiply High Word Unsigned) instruction, `mulli` or `mull` (Multiply Low Immediate) instruction, `mulw` or `muls` (Multiply Low Word) instruction.

**Fixed-Point Processor**

**Fixed-Point Arithmetic Instructions**

**Using Milicore Routines**

---

### mulhd (Multiply High Double Word) Instruction

**Purpose**

Multiply two 64-bit values together. Place the high-order 64 bits of the result into a register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>22-30</td>
<td>73</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
Description
The 64-bit operands are the contents of general purpose registers (GPR) RA and RB. The high-order 64 bits of the 128-bit product of the operands are placed into RT.

Both the operands and the product are interpreted as signed integers.

This instruction may execute faster on some implementations if RB contains the operand having the smaller absolute value.

Parameters

RT  Specifies target general-purpose register for the result of the computation.
RA  Specifies source general-purpose register for an operand.
RB  Specifies source general-purpose register for an operand.

Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

mulhdu (Multiply High Double Word Unsigned) Instruction

Purpose
Multiply 2 unsigned 64-bit values together. Place the high-order 64 bits of the result into a register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>22-30</td>
<td>9</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

Description
Both the operands and the product are interpreted as unsigned integers, except that if Rc = 1 (the mulhw. instruction) the first three bits of the condition register 0 field are set by signed comparison of the result to zero.
The 64-bit operands are the contents of *RA* and *RB*. The low-order 64 bits of the 128-bit product of the operands are placed into *RT*.

Other registers altered:

- Condition Register (CR0 field):
  - Affected: LT, GT, EQ, SO (if Rc = 1)
  - Note: The setting of CR0 bits LT, GT, and EQ is mode-dependent, and reflects overflow of the 64-bit result.

This instruction may execute faster on some implementations if *RB* contains the operand having the smaller absolute value.

**Parameters**

*RT*  Specifies target general-purpose register for the result of the computation.

*RA*  Specifies source general-purpose register for the multiplicand.

*RB*  Specifies source general-purpose register for the multiplier.

**Implementation**

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

**mulhw (Multiply High Word) Instruction**

**Purpose**

Computes the most significant 32 bits of the 64-bit product of two 32-bit integers.

**Note:** The *mulhw* instruction is supported only in the PowerPC architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td>/</td>
</tr>
<tr>
<td>22-30</td>
<td>75</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**PowerPC**

```
mulhw       RT RA RB
mulhw.     RT RA RB
```

**Description**

The *mulhw* instruction multiplies the contents of general-purpose register (GPR) *RA* and GPR *RB* and places the most significant 32 bits of the 64-bit product in the target GPR *RT*. Both the operands and the product are interpreted as signed integers.
The `mulhw` instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mulhw</code></td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>mulhw</code></td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

If the syntax form sets the Record (Rc) bit to 1, then the Less Than (LT) zero, Greater Than (GT) zero and Equal To (EQ) zero bits in Condition Register Field 0 reflect the result placed in GPR `RT`, and the Summary Overflow (SO) bit is copied from the XER to the SO bit in Condition Register Field 0.

**Parameters**

- **RT** Specifies target general-purpose register where the result of operation is stored.
- **RA** Specifies source general-purpose register for EA calculation.
- **RB** Specifies source general-purpose register for EA calculation.

**Examples**

1. The following code multiplies the contents of GPR 4 by the contents of GPR 10 and stores the result in GPR 6:

   ```
   # Assume GPR 4 contains 0x0000 0003.
   # Assume GPR 10 contains 0x0000 0002.
   mulhw 6,4,10
   # GPR 6 now contains 0x0000 0000.
   ```

2. The following code multiplies the contents of GPR 4 by the contents of GPR 10, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x0000 4500.
   # Assume GPR 10 contains 0x8000 7000.
   # Assume XER(SO) = 0.
   mulhw 6,4,10
   # GPR 6 now contains 0xFFFF DD80.
   # Condition Register Field 0 now contains 0x4.
   ```

**Related Information**

The `mul` (Multiply) instruction, `mulhwu` (Multiply High Word Unsigned) instruction, `mulli` or `muli` (Multiply Low Immediate) instruction, `mullw` or `muls` (Multiply Low Word) instruction.

**Fixed-Point Processor**

**Fixed-Point Arithmetic Instructions**

**mulhwu (Multiply High Word Unsigned) Instruction**

**Purpose**

Computes the most significant 32 bits of the 64-bit product of two unsigned 32-bit integers.

**Note:** The `mulhwu` instruction is supported only in the PowerPC architecture.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td>/</td>
</tr>
<tr>
<td>22-30</td>
<td>11</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC
mulhwu  \( RT, RA, RB \)
mulhwu. \( RT, RA, RB \)

Description
The \texttt{mulhwu} instruction multiplies the contents of general-purpose register (GPR) \( RA \) and GPR \( RB \) and places the most significant 32 bits of the 64-bit product in the target GPR \( RT \). Both the operands and the product are interpreted as unsigned integers.

\textbf{Note:} Although the operation treats the result as an unsigned integer, the setting of the Condition Register Field 0 for the Less Than (LT) zero, Greater Than (GT) zero, and Equal To (EQ) zero bits are interpreted as signed integers.

The \texttt{mulhwu} instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulhwu</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>mulhwu.</td>
<td>1</td>
<td>LT, GT, EQ, SO</td>
</tr>
</tbody>
</table>

If the syntax form sets the Record (Rc) bit to 1, then the Less Than (LT) zero, Greater Than (GT) zero and Equal To (EQ) zero bits in Condition Register Field 0 reflect the result placed in GPR \( RT \), and the Summary Overflow (SO) bit is copied from the XER to the SO bit in Condition Register Field 0.

Parameters

\( RT \) Specifies target general-purpose register where result of operation is stored.
\( RA \) Specifies source general-purpose register for EA calculation.
\( RB \) Specifies source general-purpose register for EA calculation.

Examples
1. The following code multiplies the contents of GPR 4 by the contents of GPR 10 and stores the result in GPR 6:

# Assume GPR 4 contain 0x0000 0003.
# Assume GPR 10 contains 0x0000 0002.
mulhwu 6, 4, 10
# GPR 6 now contains 0x0000 0000.
2. The following code multiplies the contents of GPR 4 by the contents of GPR 10, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x00004500.
# Assume GPR 10 contains 0x80007000.
# Assume XER(SO) = 0.
mulhw. 6,4,10
# GPR 6 now contains 0x00002280.
# Condition Register Field 0 now contains 0x4.
```

### Related Information

The **mul** (Multiply) instruction, **mulhw** (Multiply High Word) instruction, **mull** or **muli** (Multiply Low Immediate) instruction, **mullw** or **muls** (Multiply Low Word) instruction.

**Fixed-Point Processor**

**Fixed-Point Arithmetic Instructions**

### mulld (Multiply Low Double Word) Instruction

#### Purpose

Multiply 2 64-bit values together. Place the low-order 64 bits of the result into a register.

#### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>D</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>233</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**POWER family**

- mulld $RT$, $RA$, $RB$ (OE=0 Rc=0)
- mulld $RT$, $RA$, $RB$ (OE=0 Rc=1)
- mulldo $RT$, $RA$, $RB$ (OE=1 Rc=0)
- mulldo $RT$, $RA$, $RB$ (OE=1 Rc=1)

#### Description

The 64-bit operands are the contents of general purpose registers (GPR) $RA$ and $RB$. The low-order 64 bits of the 128-bit product of the operands are placed into $RT$.

Both the operands and the product are interpreted as signed integers. The low-order 64 bits of the product are independent of whether the operands are regarded as signed or unsigned 64-bit integers. If $OE = 1$ (the *mulld* and *mulldo* instructions), then OV is set if the product cannot be represented in 64 bits.

This instruction may execute faster on some implementations if $RB$ contains the operand having the smaller absolute value.

Other registers altered:
Condition Register (CR0 field):
Affected: LT, GT, EQ, SO (if Rc = 1)
Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).

XER:
Affected: SO, OV (if OE = 1)
Note: The setting of the affected bits in the XER is mode-independent, and reflects overflow of the 64-bit result.

Parameters

| RT | Specifies target general-purpose register for the result of the computation. |
| RA | Specifies source general-purpose register for an operand. |
| RB | Specifies source general-purpose register for an operand. |

Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

mulli or muli (Multiply Low Immediate) Instruction

Purpose
Multiplies the contents of a general-purpose register by a 16-bit signed integer and stores the result in another general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>07</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>SI</td>
</tr>
</tbody>
</table>

PowerPC
mulli \( RT \ [RA] \ [SI] \)

POWER family
muli \( RT \ [RA] \ [SI] \)

Description
The mulli and muli instructions sign extend the SI field to 32 bits and then multiply the extended value by the contents of general-purpose register (GPR) RA. The least significant 32 bits of the 64-bit product are placed in the target GPR RT.

The mulli and muli instructions have one syntax form and do not affect Condition Register Field 0 or the Fixed-Point Exception Register.
Parameters

\( RT \)  Specifies target general-purpose register where result of operation is stored.
\( RA \)  Specifies source general-purpose register for operation.
\( SI \)  Specifies 16-bit signed integer for operation.

Examples

The following code multiplies the contents of GPR 4 by 10 and places the result in GPR 6:

```plaintext
# Assume GPR 4 holds 0x0000 3000.
mulli 6,4,10
# GPR 6 now holds 0x0001 E000.
```

Related Information

The \texttt{mul} (Multiply) instruction, \texttt{mulhw} (Multiply High Word) instruction, \texttt{mulhwu} (Multiply High Word Unsigned) instruction, \texttt{mullw} or \texttt{muls} (Multiply Low Word) instruction.

Fixed-Point Processor

Fixed-Point Arithmetic Instructions

\texttt{mullw} or \texttt{muls} (Multiply Low Word) Instruction

Purpose

Computes the least significant 32 bits of the 64-bit product of two 32-bit integers.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>( RT )</td>
</tr>
<tr>
<td>11-15</td>
<td>( RA )</td>
</tr>
<tr>
<td>16-20</td>
<td>( RB )</td>
</tr>
<tr>
<td>21</td>
<td>( OE )</td>
</tr>
<tr>
<td>22-30</td>
<td>235</td>
</tr>
<tr>
<td>31</td>
<td>( Rc )</td>
</tr>
</tbody>
</table>

\textbf{PowerPC}

\texttt{mullw} \( RT \ \ RA \ \ RB \)
\texttt{mullw.} \( RT \ \ RA \ \ RB \)
\texttt{mullwo} \( RT \ \ RA \ \ RB \)
\texttt{mullwo.} \( RT \ \ RA \ \ RB \)

\textbf{POWER family}

\texttt{muls} \( RT \ \ RA \ \ RB \)
\texttt{muls.} \( RT \ \ RA \ \ RB \)
\texttt{mulso} \( RT \ \ RA \ \ RB \)
\texttt{mulso.} \( RT \ \ RA \ \ RB \)
Description

The **mullw** and **muls** instructions multiply the contents of general-purpose register (GPR) \( RA \) by the contents of GPR \( RB \), and place the least significant 32 bits of the result in the target GPR \( RT \).

The **mullw** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

The **muls** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>mullw</td>
<td>0</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>mullw.</td>
<td>0</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ</td>
</tr>
<tr>
<td>mullwo</td>
<td>1</td>
<td>SO,OV</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>mullwo.</td>
<td>1</td>
<td>SO,OV</td>
<td>1</td>
<td>LT,GT,EQ</td>
</tr>
<tr>
<td>muls</td>
<td>0</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>muls.</td>
<td>0</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ</td>
</tr>
<tr>
<td>mulso</td>
<td>1</td>
<td>SO,OV</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>mulso.</td>
<td>1</td>
<td>SO,OV</td>
<td>1</td>
<td>LT,GT,EQ</td>
</tr>
</tbody>
</table>

The four syntax forms of the **mullw** instruction, and the four syntax forms of the **muls** instruction, never affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction sets the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register to 1 if the result is too large to be represented in 32 bits. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

- \( RT \) Specifies target general-purpose register where result of operation is stored.
- \( RA \) Specifies source general-purpose register for operation.
- \( RB \) Specifies source general-purpose register for operation.

Examples

1. The following code multiplies the contents of GPR 4 by the contents of GPR 10 and stores the result in GPR 6:
   ```assembly
   # Assume GPR 4 holds 0x0000 3000.
   # Assume GPR 10 holds 0x0000 7000.
   mullw 6,4,10
   # GPR 6 now holds 0x1500 0000.
   ```

2. The following code multiplies the contents of GPR 4 by the contents of GPR 10, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:
   ```assembly
   # Assume GPR 4 holds 0x0000 4500.
   # Assume GPR 10 holds 0x0000 7000.
   # Assume XER(SO) = 0.
   mullw. 6,4,10
   # GPR 6 now holds 0x1E30 0000.
   # Condition Register Field 0 now contains 0x4.
   ```
3. The following code multiplies the contents of GPR 4 by the contents of GPR 10, stores the result in GPR 6, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register to reflect the result of the operation:

```c
# Assume GPR 4 holds 0x0000 4500.
# Assume GPR 10 holds 0x0007 0000.
# Assume XER = 0.
mullwo 6,4,10
# GPR 6 now holds 0xE300 0000.
# XER now contains 0xc000 0000
```

4. The following code multiplies the contents of GPR 4 by the contents of GPR 10, stores the result in GPR 6, and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```c
# Assume GPR 4 holds 0x0000 4500.
# Assume GPR 10 holds 0x7FFF FFFF.
# Assume XER = 0.
mullwo. 6,4,10
# GPR 6 now holds 0xFFFF BB00.
# XER now contains 0xc000 0000
# Condition Register Field 0 now contains 0x9.
```

**Related Information**

The `mul` (Multiply) instruction, `mulhw` (Multiply High Word) instruction, `mulhwu` (Multiply High Word Unsigned) instruction, `mulh` or `muli` (Multiply Low Immediate) instruction.

**Fixed-Point Processor**

**Fixed-Point Arithmetic Instructions**

---

**nabs (Negative Absolute) Instruction**

**Purpose**

Negates the absolute value of the contents of a general-purpose register and stores the result in another general-purpose register.

**Note:** The `nabs` instruction is supported only in the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>488</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**POWER family**

- `nabs` | `RT RA`
- `nabs` | `RT RA`
- `nabso` | `RT RA`
- `nabso` | `RT RA`
Description
The `nabs` instruction places the negative absolute value of the contents of general-purpose register (GPR) \( RA \) into the target GPR \( RT \).

The `nabs` instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>nabs</td>
<td>0</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>nabs.</td>
<td>0</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>nabso</td>
<td>1</td>
<td>SO,OV</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>nabso.</td>
<td>1</td>
<td>SO,OV</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The four syntax forms of the `nabs` instruction never affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the Summary Overflow (SO) bit is unchanged and the Overflow (OV) bit is set to zero. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters
- \( RT \) Specifies target general-purpose register where result of operation is stored.
- \( RA \) Specifies source general-purpose register for operation.

Examples
1. The following code takes the negative absolute value of the contents of GPR 4 and stores the result in GPR 6:
   ```
   # Assume GPR 4 contains 0x0000 3000.
   nabs 6,4
   # GPR 6 now contains 0xFFFF D000.
   ```
2. The following code takes the negative absolute value of the contents of GPR 4, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:
   ```
   # Assume GPR 4 contains 0xFFFF FFFF.
   nabs. 6,4
   # GPR 6 now contains 0xFFFF FFFF.
   ```
3. The following code takes the negative absolute value of the contents of GPR 4, stores the result in GPR 6, and sets the Overflow bit in the Fixed-Point Exception Register to 0:
   ```
   # Assume GPR 4 contains 0x0000 0001.
   nabso 6,4
   # GPR 6 now contains 0xFFFF FFFF.
   ```
4. The following code takes the negative absolute value of the contents of GPR 4, stores the result in GPR 6, sets Condition Register Field 0 to reflect the result of the operation, and sets the Overflow bit in the Fixed-Point Exception Register to 0:
   ```
   # Assume GPR 4 contains 0x8000 0000.
   nabso 6,4
   # GPR 6 now contains 0x8000 0000.
   ```
nand (NAND) Instruction

Purpose
Logically complements the result of ANDing the contents of two general-purpose registers and stores the result in another general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>476</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

nand(RA, RS, RB)
nand(RA, RS, RB)

Description
The nand instruction logically ANDs the contents of general-purpose register (GPR) RS with the contents of GPR RB and stores the complement of the result in the target GPR RA.

The nand instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>nand</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>nand.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the nand instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters
RA \( \) Specifies target general-purpose register where result of operation is stored.
RS \( \) Specifies source general-purpose register for operation.
RB \( \) Specifies source general-purpose register for operation.
Examples

1. The following code complements the result of ANDing the contents of GPR 4 and GPR 7 and stores the result in GPR 6:

   ```
   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 7 contains 0x789A 789B.
   nand 6,4,7
   # GPR 6 now contains 0xEFFF CFFF.
   ```

2. The following code complements the result of ANDing the contents of GPR 4 and GPR 7, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0xB004 3000.
   # Assume GPR 7 contains 0x789A 789B.
   nand. 6,4,7
   # GPR 6 now contains 0xCFFF CFFF.
   ```

Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Logical Instructions](#)

**neg (Negate) Instruction**

**Purpose**

Changes the arithmetic sign of the contents of a general-purpose register and places the result in another general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>104</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**Description**

The `neg` instruction adds 1 to the one's complement of the contents of a general-purpose register (GPR) `RA` and stores the result in GPR `RT`.

If GPR `RA` contains the most negative number (that is, 0x8000 0000), the result of the instruction is the most negative number and signals the Overflow bit in the Fixed-Point Exception Register if OE is 1.

The `neg` instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.
The four syntax forms of the `neg` instruction never affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

### Parameters

- **RT**  
  Specifies target general-purpose register where result of operation is stored.
- **RA**  
  Specifies source general-purpose register for operation.

### Examples

1. The following code negates the contents of GPR 4 and stores the result in GPR 6:
   
   ```
   # Assume GPR 4 contains 0x9000 3000.
   neg 6,4
   # GPR 6 now contains 0x6FFF D000.
   ```

2. The following code negates the contents of GPR 4, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:
   
   ```
   # Assume GPR 4 contains 0x789A 789B.
   neg. 6,4
   # GPR 6 now contains 0x8765 8765.
   ```

3. The following code negates the contents of GPR 4, stores the result in GPR 6, and sets the Fixed-Point Exception Register Summary Overflow and Overflow bits to reflect the result of the operation:
   
   ```
   # Assume GPR 4 contains 0x9000 3000.
   nego 6,4
   # GPR 6 now contains 0x6FFF D000.
   ```

4. The following code negates the contents of GPR 4, stores the result in GPR 6, and sets Condition Register Field 0 and the Fixed-Point Exception Register Summary Overflow and Overflow bits to reflect the result of the operation:
   
   ```
   # Assume GPR 4 contains 0x8000 0000.
   nego. 6,4
   # GPR 6 now contains 0x8000 0000.
   ```

### Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Arithmetic Instructions](#)
nor (NOR) Instruction

Purpose
Logically complements the result of ORing the contents of two general-purpose registers and stores the result in another general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>124</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

nor.  
or.

See Extended Mnemonics of Fixed-Point Logical Instructions for more information.

Description
The nor instruction logically ORs the contents of general-purpose register (GPR) RS with the contents of GPR RB and stores the complemented result in GPR RA.

The nor instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>nor</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>nor.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the nor instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters
RA Specifies target general-purpose register where result of operation is stored.
RS Specifies source general-purpose register for operation.
RB Specifies source general-purpose register for operation.

Examples
1. The following code NORs the contents of GPR 4 and GPR 7 and stores the result in GPR 6:
2. The following code NORs the contents of GPR 4 and GPR 7, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x8000 3000.
# Assume GPR 7 contains 0x789A 789B.
nor 6,4,7
# GPR 6 now contains 0x0761 8764.
```

### Related Information
- [Fixed-Point Processor](#)
- [Fixed-Point Logical Instructions](#)

### or (OR) Instruction

#### Purpose
Logically ORs the contents of two general-purpose registers and stores the result in another general-purpose register.

#### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>444</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

```
or       RA RS RB
or.      RA RS RB
```

See [Extended Mnemonics of Fixed-Point Logical Instructions](#) for more information.

#### Description
The `or` instruction logically ORs the contents of general-purpose register (GPR) `RS` with the contents of GPR `RB` and stores the result in GPR `RA`.

The `or` instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>or.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>
The two syntax forms of the or instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA  Specifies target general-purpose register where result of operation is stored.
RS  Specifies source general-purpose register for operation.
RB  Specifies source general-purpose register for operation.

Examples

1. The following code logically ORs the contents of GPR 4 and GPR 7 and stores the result in GPR 6:

   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 7 contains 0x789A 789B.
   or 6,4,7
   # GPR 6 now contains 0xF89A 789B.

2. The following code logically ORs the contents of GPR 4 and GPR 7, loads the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   # Assume GPR 4 contains 0xB004 3000.
   # Assume GPR 7 contains 0x789A 789B.
   or. 6,4,7
   # GPR 6 now contains 0xF89E 789B.

Related Information

Fixed-Point Processor.

Fixed-Point Logical Instructions.

orc (OR with Complement) Instruction

Purpose

Logically ORs the contents of a general-purpose register with the complement of the contents of another general-purpose register and stores the result in a third general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>412</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

orc    RA RS RB
orc.   RA RS RB
Description

The orc instruction logically ORs the contents of general-purpose register (GPR) RS with the complement of the contents of GPR RB and stores the result in GPR RA.

The orc instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>orc</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>orc.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the orc instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general-purpose register where result of operation is stored.
RS Specifies source general-purpose register for operation.
RB Specifies source general-purpose register for operation.

Examples

1. The following code logically ORs the contents of GPR 4 with the complement of the contents of GPR 7 and stores the result in GPR 6:

   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 7 contains 0x789A 789B, whose complement is 0x8765 8764.
   orc 6,4,7
   # GPR 6 now contains 0x9765 B764.

2. The following code logically ORs the contents of GPR 4 with the complement of the contents GPR 7, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   # Assume GPR 4 contains 0x8004 3000.
   # Assume GPR 7 contains 0x789A 789B, whose complement is 0x8765 8764.
   orc 6,4,7
   # GPR 6 now contains 0x8765 B764.

Related Information

- Fixed-Point Processor
- Fixed-Point Logical Instructions

ori or oril (OR Immediate) Instruction

Purpose

Logically ORs the lower 16 bits of the contents of a general-purpose register with a 16-bit unsigned integer and stores the result in another general-purpose register.
### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>24</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>UI</td>
</tr>
</tbody>
</table>

#### PowerPC
ori

#### POWER family
oril

See [Extended Mnemonics of Fixed-Point Logical Instructions](#) for more information.

### Description

The **ori** and **oril** instructions logically OR the contents of general-purpose register (GPR) **RS** with the concatenation of x'0000' and a 16-bit unsigned integer, **UI**, and place the result in GPR **RA**.

The **ori** and **oril** instructions have one syntax form and do not affect Condition Register Field 0 or the Fixed-Point Exception Register.

### Parameters

- **RA** Specifies target general-purpose register where result of operation is stored.
- **RS** Specifies source general-purpose register for operation.
- **UI** Specifies a 16-bit unsigned integer for operation.

### Examples

The following code ORs the lower 16 bits of the contents of GPR 4 with 0x0079 and stores the result in GPR 6:

```plaintext
# Assume GPR 4 contains 0x9000 3000.
ori 6,4,0x0079
# GPR 6 now contains 0x9000 3079.
```

### Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Logical Instructions](#)

### oris or oriu (OR Immediate Shifted) Instruction

#### Purpose

Logically ORs the upper 16 bits of the contents of a general-purpose register with a 16-bit unsigned integer and stores the result in another general-purpose register.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>25</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>UI</td>
</tr>
</tbody>
</table>

PowerPC
oris RA RS UI

POWER family
oriu RA RS UI

Description
The `oris` and `oriu` instructions logically OR the contents of general-purpose register (GPR) `RS` with the concatenation of a 16-bit unsigned integer, `UI`, and x’0000’ and store the result in GPR `RA`.

The `oris` and `oriu` instructions have one syntax form and do not affect Condition Register Field 0 or the Fixed-Point Exception Register.

Parameters

`RA`  Specifies target general-purpose register where result of operation is stored.

`RS`  Specifies source general-purpose register for operation.

`UI`  Specifies a 16-bit unsigned integer for operation.

Examples
The following code ORs the upper 16 bits of the contents of GPR 4 with 0x0079 and stores the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
oris 6,4,0x0079
# GPR 6 now contains 0x9079 3000.
```

Related Information

- Fixed-Point Processor
- Fixed-Point Logical Instructions

**popcntbd (Population Count Byte Doubleword) Instruction**

**Purpose**
Allows a program to count the number of one bits in a doubleword.

**Note:** The `popcntbd` instruction is supported for POWER5 architecture only.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21–30</td>
<td>122</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

**POWER5**

**popcntbd** “RS” “RA”

**Description**
The `popcntbd` instruction counts the number of one bits in each byte of register `RS` and places the count in to the corresponding byte of register `RA`. The number ranges from 0 to 8, inclusive.

The `popcntbd` instruction has one syntax form and does not affect any Special Registers.

**Parameters**

- **RS**  Specifications for source general-purpose register.
- **RA**  Specifications for destination general-purpose register.

**Related Information**

“cntlzw or cntlz (Count Leading Zeros Word) Instruction” on page 162.

**rac (Real Address Compute) Instruction**

**Purpose**
Translates an effective address into a real address and stores the result in a general-purpose register.

**Note:** The `rac` instruction is supported only in the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>818</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
Description
The rac instruction computes an effective address (EA) from the sum of the contents of general-purpose register (GPR) RA and the contents of GPR RB, and expands the EA into a virtual address.

If RA is not 0 and if RA is not RT, then the rac instruction stores the EA in GPR RA, translates the result into a real address, and stores the real address in GPR RT.

Consider the following when using the rac instruction:
- If GPR RA is 0, then EA is the sum of the contents of GPR RB and 0.
- EA is expanded into its virtual address and translated into a real address, regardless of whether data translation is enabled.
- If the translation is successful, the EQ bit in the condition register is set and the real address is placed in GPR RT.
- If the translation is unsuccessful, the EQ bit is set to 0, and 0 is placed in GPR RT.
- If the effective address specifies an I/O address, the EQ bit is set to 0, and 0 is placed in GPR RT.
- The reference bit is set if the real address is not in the Translation Look-Aside buffer (TLB).

The rac instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rac</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>rac</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the rac instruction do not affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction effects the Equal (EQ) and Summary Overflow (SO) bit in Condition Register Field 0.

Note: The hardware may first search the Translation Look-Aside buffer for the address. If this fails, the Page Frame table must be searched. In this case, it is not necessary to load a Translation Look-Aside buffer entry.

Parameters

- **RT** Specifies the target general-purpose register where result of operation is stored.
- **RA** Specifies the source general-purpose register for EA calculation.
- **RB** Specifies the source general-purpose register for EA calculation.

Security
The rac instruction instruction is privileged.

Related Information
Processing and Storage
rfi (Return from Interrupt) Instruction

**Purpose**
Reinitializes the Machine State Register and continues processing after an interrupt.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>19</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>50</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

rfi

**Description**
The rfi instruction places bits 16-31 of Save Restore Register1 (SRR1) into bits 16-31 of the Machine State Register (MSR), and then begins fetching and processing instructions at the address contained in Save Restore Register0 (SRR0), using the new MSR value.

If the Link bit (LK) is set to 1, the contents of the Link Register are undefined.

The rfi instruction has one syntax form and does not affect Condition Register Field 0 or the Fixed-Point Exception Register.

**Security**
The rfi instruction is privileged and synchronizing.

**Related Information**
Branch Processor.

rfid (Return from Interrupt Double Word) Instruction

**Purpose**
Reinitializes the Machine State Register and continues processing after an interrupt.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>19</td>
</tr>
<tr>
<td>6-10</td>
<td>00000</td>
</tr>
<tr>
<td>11-15</td>
<td>00000</td>
</tr>
<tr>
<td>16-20</td>
<td>00000</td>
</tr>
<tr>
<td>21-30</td>
<td>18</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>
rfid

Description
Bits 0, 48-55, 57-59, and 62-63 from the Save Restore Register 1 (SRR1) are placed into the corresponding bits of the Machine State Register (MSR). If the new MSR value does not enable any pending exceptions, then the next instruction is fetched under control of the new MSR value. If the SF bit in the MSR is 1, the address found in bits 0-61 of SRR0 (fullword aligned address) becomes the next instruction address. If the SF bit is zero, then bits 32-61 of SRR0, concatenated with zeros to create a word-aligned address, are placed in the low-order 32-bits of SRR0. The high-order 32 bits are cleared. If the new MSR value enables one or more pending exceptions, the exception associated with the highest priority pending exception is generated; in this case the value placed into SRR0 by the exception processing mechanism is the address of the instruction that would have been executed next had the exception not occurred.

Other registers altered:
• MSR

Security
The rfid instruction is privileged and synchronizing.

Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation causes an illegal instruction type program exception.

rfsvc (Return from SVC) Instruction

Purpose
Reinitializes the Machine State Register and starts processing after a supervisor call (svc).

Note: The rfsvc instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>19</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>82</td>
</tr>
<tr>
<td>31</td>
<td>LK</td>
</tr>
</tbody>
</table>

POWER family

rfsvc
Description
The `rfsvc` instruction reinitializes the Machine State Register (MSR) and starts processing after a supervisor call. This instruction places bits 16-31 of the Count Register into bits 16-31 of the Machine State Register (MSR), and then begins fetching and processing instructions at the address contained in the Link Register, using the new MSR value.

If the Link bit (LK) is set to 1, then the contents of the Link Register are undefined.

The `rfsvc` instruction has one syntax form and does not affect Condition Register Field 0 or the Fixed-Point Exception Register.

Security
The `rfsvc` instruction is privileged and synchronizing.

Related Information
The `svc` (Supervisor Call) instruction.

Branch Processor.

System Call Instructions.

**rldcl (Rotate Left Double Word then Clear Left) Instruction**

Purpose
Rotate the contents of a general purpose register left by the number of bits specified by the contents of another general purpose register. Generate a mask that is ANDed with the result of the shift operation. Store the result of this operation in another general purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>30</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-26</td>
<td>mb</td>
</tr>
<tr>
<td>27-30</td>
<td>8</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family
`rldcl` `RA RS RB MB` (Rc=0)
`rldcl` `RA RS RB MB` (Rc=1)

Description
The contents of general purpose register (GPR) `RS` are rotated left the number of bits specified by the operand in the low-order six bits of `RB`. A mask is generated having 1 bits from bit `MB` through bit 63 and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into `RA`.

Note that the `rldcl` instruction can be used to extract and rotate bit fields using the methods shown below:
To extract an n-bit field, that starts at variable bit position b in register \( RS \), right-justified into \( RA \) (clearing the remaining 64 - n bits of \( RA \)), set the low-order six bits of \( RB \) to \( b + n \) and \( MB = 64 - n \).

To rotate the contents of a register left by variable n bits, set the low-order six bits of \( RB \) to \( n \) and \( MB = 0 \), and to shift the contents of a register right, set the low-order six bits of \( RB \) to \( (64 - n) \) and \( MB = 0 \).

Other registers altered:
- Condition Register (CR0 field):
  - Affected: LT, GT, EQ, SO (if \( Rc = 1 \))

**Parameters**

- \( RA \) Specifies the target general purpose register for the result of the instruction.
- \( RS \) Specifies the source general purpose register containing the operand.
- \( RB \) Specifies the source general purpose register containing the shift value.
- \( MB \) Specifies the begin value (bit number) of the mask for the operation.

**Implementation**

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

**rldicl (Rotate Left Double Word Immediate then Clear Left) Instruction**

**Purpose**

This instruction should only be used on 64-bit PowerPC processors running a 64-bit application.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>30</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>sh</td>
</tr>
<tr>
<td>21-26</td>
<td>mb</td>
</tr>
<tr>
<td>27-29</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>sh</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**Description**

The contents of \( RS \) are rotated left the number of bits specified by operand \( SH \). A mask is generated having 1 bits from bit \( MB \) through bit 63 and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into \( RA \).

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.
Note that rldcl can be used to extract, rotate, shift, and clear bit fields using the methods shown below:

To extract an n-bit field, that starts at bit position b in rS, right-justified into rA (clearing the remaining 64 - n bits of rA), set SH = b + n and MB = 64 - n.

To rotate the contents of a register left by n bits, set SH = n and MB = 0; to rotate the contents of a register right by n bits, set SH = (64 - n), and MB = 0.

To shift the contents of a register right by n bits, set SH = 64 - n and MB = n.

To clear the high-order n bits of a register, set SH = 0 and MB = n.

Other registers altered:
- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

**Parameters**

<table>
<thead>
<tr>
<th>rA <em><strong>DESCRIPTION</strong></em></th>
<th>rS <em><strong>DESCRIPTION</strong></em></th>
<th>rB <em><strong>DESCRIPTION</strong></em></th>
<th>MB <em><strong>DESCRIPTION</strong></em></th>
</tr>
</thead>
</table>

**Examples**

**Related Information**

**rldcr (Rotate Left Double Word then Clear Right) Instruction**

**Purpose**

Rotate the contents of a general purpose register left by the number of bits specified by the contents of another general purpose register. Generate a mask that is ANDed with the result of the shift operation. Store the result of this operation in another general purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>30</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-26</td>
<td>me</td>
</tr>
<tr>
<td>27-30</td>
<td>9</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**POWER family**

rldcr       RA| RS| RB| ME (Rc=0)
rldcr.     RA| RS| RB| ME (Rc=1)
Description
The contents of general purpose register (GPR) $RS$ are rotated left the number of bits specified by the low-order six bits of $RB$. A mask is generated having 1 bits from bit 0 through bit $ME$ and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into $RA$.

Note that rldcr can be used to extract and rotate bit fields using the methods shown below:

- To extract an n-bit field, that starts at variable bit position $b$ in register $RS$, left-justified into $RA$ (clearing the remaining 64 - $n$ bits of $RA$), set the low-order six bits of $RB$ to $b$ and $ME = n - 1$.
- To rotate the contents of a register left by variable $n$ bits, set the low-order six bits of $RB$ to $n$ and $ME = 63$, and to shift the contents of a register right, set the low-order six bits of $RB$ to(64 - $n$), and $ME = 63$.

Other registers altered:
- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if $Rc = 1$)

Parameters

- $RS$ SH Specifies shift value for operation.
- $MB$ Specifies begin value of mask for operation.
- $ME$ BM Specifies value of 32-bit mask
- $RA$ Specifies target general-purpose register where result of operation is stored.
- $RS$ Specifies source general-purpose register for operation.
- $RB$ Specifies the source general-purpose register containing the shift value.
- $ME$ Specifies end value of mask for operation.

Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

rldic (Rotate Left Double Word Immediate then Clear) Instruction

Purpose
The contents of a general purpose register are rotated left a specified number of bits, then masked with a bit-field to clear some number of low-order and high-order bits. The result is placed in another general purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>30</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>sh</td>
</tr>
<tr>
<td>21-26</td>
<td>mb</td>
</tr>
<tr>
<td>27-29</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>sh</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
POWER family

rldicl\((Rc=0)\)
rldicl\((Rc=1)\)

Description
The contents of general purpose register (GPR) \(RS\) are rotated left the number of bits specified by operand \(SH\). A mask is generated having 1 bits from bit \(MB\) through bit 63 - \(SH\) and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into GPR \(RA\).

Note that \texttt{rldic} can be used to clear and shift bit fields using the methods shown below:

- To clear the high-order \(b\) bits of the contents of a register and then shift the result left by \(n\) bits, set \(SH = n\) and \(MB = b - n\).
- To clear the high-order \(n\) bits of a register, set \(SH = 0\) and \(MB = n\).

Other registers altered:
- Condition Register (CR0 field):
  - Affected: LT, GT, EQ, SO (if \(Rc = 1\))

Parameters
\(RA\) Specifies the target general purpose register for the result of the instruction.
\(RS\) Specifies the source general purpose register containing the operand.
\(SH\) Specifies the (immediate) shift value for the operation.
\(MB\) Specifies the begin value of the bit-mask for the operation.

Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

\textbf{rldicl (Rotate Left Double Word Immediate then Clear Left) Instruction}

Purpose
Rotate the contents of a general purpose register left by a specified number of bits, clearing a specified number of high-order bits. The result is placed in another general purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>30</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>sh</td>
</tr>
<tr>
<td>21-26</td>
<td>mb</td>
</tr>
<tr>
<td>27-29</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>sh</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
Description
The contents of general purpose register RS are rotated left the number of bits specified by operand SH. A mask is generated containing 1 bits from bit MB through bit 63 and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into GPR RA.

Note that rldicr can be used to extract, rotate, shift, and clear bit fields using the methods shown below:
- To extract an n-bit field, which starts at bit position b in RS, right-justified into GPR RA (clearing the remaining 64 - n bits of GPR RA), set SH = b + n and MB = 64 - n.
- To rotate the contents of a register left by n bits, set SH = n and MB = 0; to rotate the contents of a register right by n bits, set SH = (64 - n), and MB = 0.
- To shift the contents of a register right by n bits, set SH = 64 - n and MB = n.
- To clear the high-order n bits of a register, set SH = 0 and MB = n.

Other registers altered:
- Condition Register (CR0 field):
  - Affected: LT, GT, EQ, SO (if Rc = 1)

Parameters
- RA: Specifies the target general purpose register for the result of the instruction.
- RS: Specifies the source general purpose register containing the operand.
- SH: Specifies the (immediate) shift value for the operation.
- MB: Specifies the begin value (bit number) of the mask for the operation.

Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

rldicr (Rotate Left Double Word Immediate then Clear Right)

Purpose
Rotate the contents of a general purpose register left by the number of bits specified by an immediate value. Clear a specified number of low-order bits. Place the results in another general purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>30</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>sh</td>
</tr>
<tr>
<td>21-26</td>
<td>me</td>
</tr>
</tbody>
</table>
Bits | Value
---|---
27-29 | 1
30 | sh
31 | Rc

POWER family
**rldicr**  
RA RS SH MB (Rc=0)
**rldicr.**  
RA RS SH MB (Rc=1)

**Description**
The contents of general purpose register (GPR) Rs are rotated left the number of bits specified by operand SH. A mask is generated having 1 bits from bit 0 through bit ME and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into GPR RA.

Note that rldicr can be used to extract, rotate, shift, and clear bit fields using the methods shown below:

- To extract an n-bit field, that starts at bit position b in GPR RS, left-justified into GPR RA (clearing the remaining 64 - n bits of GPR RA), set SH = b and ME = n - 1.
- To rotate the contents of a register left (right) by n bits, set SH = n (64 - n) and ME = 63.
- To shift the contents of a register left by n bits, by setting SH = n and ME = 63 - n.
- To clear the low-order n bits of a register, by setting SH = 0 and ME = 63 - n.

Other registers altered:
- Condition Register (CR0 field):
  - Affected: LT, GT, EQ, SO (if Rc = 1)

**Parameters**

*RA*  
Specifies the target general purpose register for the result of the instruction.

*RS*  
Specifies the source general purpose register containing the operand.

*SH*  
Specifies the (immediate) shift value for the operation.

*ME*  
Specifies the end value (bit number) of the mask for the operation.

**Implementation**
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

**rldimi (Rotate Left Double Word Immediate then Mask Insert)**

**Instruction**

**Purpose**
The contents of a general purpose register are rotated left a specified number of bits. A generated mask is used to insert a specified bit-field into the corresponding bit-field of another general purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>30</td>
</tr>
<tr>
<td>Bits</td>
<td>Value</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>sh</td>
</tr>
<tr>
<td>2126</td>
<td>mb</td>
</tr>
<tr>
<td>27-29</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>sh</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**POWER family**

**rldimi**  
\[ RA \quad RS \quad SH \quad MB \] (Rc=0)

**rldimi.**  
\[ RA \quad RS \quad SH \quad MB \] (Rc=1)

**Description**
The contents of general purpose register (GPR) \( RS \) are rotated left the number of bits specified by operand \( SH \). A mask is generated having 1 bits from bit \( MB \) through bit \( 63 - SH \) and 0 bits elsewhere. The rotated data is inserted into \( RA \) under control of the generated mask.

Note that rldimi can be used to insert an n-bit field, that is right-justified in \( RS \), into \( RA \) starting at bit position \( b \), by setting \( SH = 64 - (b + n) \) and \( MB = b \).

Other registers altered:
- Condition Register (CR0 field):
  - Affected: LT, GT, EQ, SO (if Rc = 1)

**Parameters**
- \( RA \)  
  Specifies the target general purpose register for the result of the instruction.
- \( RS \)  
  Specifies the source general purpose register containing the operand.
- \( SH \)  
  Specifies the (immediate) shift value for the operation.
- \( MB \)  
  Specifies the begin value of the bit-mask for the operation.

**Implementation**
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

**rlmi (Rotate Left Then Mask Insert) Instruction**

**Purpose**
Rotates the contents of a general-purpose register to the left by the number of bits specified in another general-purpose register and stores the result in a third general-purpose register under the control of a generated mask.

**Note:** The rlmi instruction is supported only in the POWER family architecture.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>22</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-25</td>
<td>MB</td>
</tr>
<tr>
<td>26-30</td>
<td>ME</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**POWER family**

`rlmi`  
`rlmi.`

See [Extended Mnemonics of Fixed-Point Rotate and Shift Instructions](#) for more information.

**Description**

The `rlmi` instruction rotates the contents of the source general-purpose register (GPR) RS to the left by the number of bits specified by bits 27-31 of GPR RB and then stores the rotated data in GPR RA under control of a 32-bit generated mask defined by the values in Mask Begin (MB) and Mask End (ME).

Consider the following when using the `rlmi` instruction:

- If a mask bit is 1, the instruction places the associated bit of rotated data in GPR RA; if a mask bit is 0, the GPR RA bit remains unchanged.
- If the MB value is less than the ME value + 1, then the mask bits between and including the starting point and the end point are set to ones. All other bits are set to zeros.
- If the MB value is the same as the ME value + 1, then all 32 mask bits are set to ones.
- If the MB value is greater than the ME value + 1, then all of the mask bits between and including the ME value +1 and the MB value -1 are set to zeros. All other bits are set to ones.

The parameter BM can also be used to specify the mask for this instruction. The assembler will generate the MB and ME parameters from BM.

The `rlmi` instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rlmi</code></td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>rlmi.</code></td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the `rlmi` instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.
Parameters

RA  Specifies target general-purpose register where result of operation is stored.
RS  Specifies source general-purpose register for operation.
RB  Specifies general-purpose register that contains number of bits for rotation of data.
MB  Specifies begin value of mask for operation.
ME  Specifies end value of mask for operation.
BM  Specifies value of 32-bit mask.

Examples

1. The following code rotates the contents of GPR 4 by the value contained in bits 27-31 in GPR 5 and
   stores the masked result in GPR 6:
   
   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 5 contains 0x0000 0002.
   # Assume GPR 6 contains 0xFFFF FFFF.
   rlmi 6,4,5,0,0x1D
   # GPR 6 now contains 0x4000 C003.
   # Under the same conditions
   # rlmi 6,4,5,0xFFFFF
   # will produce the same result.

2. The following code rotates the contents of GPR 4 by the value contained in bits 27-31 in GPR 5,
   stores the masked result in GPR 6, and sets Condition Register Field 0 to reflect the result of the
   operation:
   
   # Assume GPR 4 contains 0xB004 3000.
   # Assume GPR 5 contains 0x0000 0002.
   # GPR 6 is the target register and contains 0xFFFF FFFF.
   rlmi. 6,4,5,0,0x1D
   # GPR 6 now contains 0xC010 C003.
   # CRF 0 now contains 0x8.
   # Under the same conditions
   # rlmi. 6,4,5,0xFFFFF
   # will produce the same result.

Related Information

Fixed-Point Processor.

Fixed-Point Rotate and Shift Instructions.

rlwimi or rlimi (Rotate Left Word Immediate Then Mask Insert) Instruction

Purpose
Rotates the contents of a general-purpose register to the left by a specified number of bits and stores the
result in another general-purpose register under the control of a generated mask.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>20</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>SH</td>
</tr>
</tbody>
</table>
### Description

The `rlwimi` and `rlimi` instructions rotate left the contents of the source general-purpose register (GPR) `RS` by the number of bits by the `SH` parameter and then store the rotated data in GPR `RA` under control of a 32-bit generated mask defined by the values in Mask Begin (`MB`) and Mask End (`ME`). If a mask bit is 1, the instructions place the associated bit of rotated data in GPR `RA`; if a mask bit is 0, the GPR `RA` bit remains unchanged.

Consider the following when using the `rlwimi` and `rlimi` instructions:

- If the `MB` value is less than the `ME` value + 1, then the mask bits between and including the starting point and the end point are set to ones. All other bits are set to zeros.
- If the `MB` value is the same as the `ME` value + 1, then all 32 mask bits are set to ones.
- If the `MB` value is greater than the `ME` value + 1, then all of the mask bits between and including the `ME` value +1 and the `MB` value -1 are set to zeros. All other bits are set to ones.

The `BM` parameter can also be used to specify the mask for these instructions. The assembler will generate the `MB` and `ME` parameters from the `BM` value.

The `rlwimi` and `rlimi` instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rlwimi</code></td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>rlwimi.</code></td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td><code>rlimi</code></td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>rlimi.</code></td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The syntax forms of the `rlwimi` and `rlimi` instructions never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instructions affect the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.
Parameters

RA  Specifies target general-purpose register where result of operation is stored.
RS  Specifies source general-purpose register for operation.
SH  Specifies shift value for operation.
MB  Specifies begin value of mask for operation.
ME  Specifies end value of mask for operation.
BM  Specifies value of 32-bit mask.

Examples

1. The following code rotates the contents of GPR 4 to the left by 2 bits and stores the masked result in GPR 6:
   
   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 6 contains 0x0000 0003.
   rlwimi 6,4,2,0,0x1D
   # GPR 6 now contains 0x4000 C003.
   # Under the same conditions
   # rlwimi 6,4,2,0xFFFFFFFC
   # will produce the same result.

2. The following code rotates the contents of GPR 4 to the left by 2 bits, stores the masked result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:
   
   # Assume GPR 4 contains 0x789A 789B.
   # Assume GPR 6 contains 0x3000 0003.
   rlwimi. 6,4,2,0,0x1A
   # GPR 6 now contains 0xE269 E263.
   # CRF 0 now contains 0x8.
   # Under the same conditions
   # rlwimi. 6,4,2,0xFFFFFFE0
   # will produce the same result.

Related Information

Fixed-Point Processor

Fixed-Point Rotate and Shift Instructions

rlwinm or rlinm (Rotate Left Word Immediate Then AND with Mask) Instruction

Purpose

Logically ANDs a generated mask with the result of rotating left by a specified number of bits in the contents of a general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>21</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>SH</td>
</tr>
<tr>
<td>21-25</td>
<td>MB</td>
</tr>
<tr>
<td>26-30</td>
<td>ME</td>
</tr>
</tbody>
</table>
### Description

The `rlwinm` and `rlinm` instructions rotate left the contents of the source general-purpose register (GPR) `RS` by the number of bits specified by the `SH` parameter, logically AND the rotated data with a 32-bit generated mask defined by the values in Mask Begin (`MB`) and Mask End (`ME`), and store the result in GPR `RA`.

Consider the following when using the `rlwinm` and `rlinm` instructions:

- If the `MB` value is less than the `ME` value + 1, then the mask bits between and including the starting point and the end point are set to ones. All other bits are set to zeros.
- If the `MB` value is the same as the `ME` value + 1, then all 32 mask bits are set to ones.
- If the `MB` value is greater than the `ME` value + 1, then all of the mask bits between and including the `ME` value +1 and the `MB` value -1 are set to zeros. All other bits are set to ones.

The `BM` parameter can also be used to specify the mask for these instructions. The assembler will generate the `MB` and `ME` parameters from the `BM` value.

The `rlwinm` and `rlinm` instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rlwinm</code></td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td><code>rlwinm</code></td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td><code>rlinm</code></td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>rlinm</code></td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The syntax forms of the `rlwinm` and `rlinm` instructions never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instructions affect the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

### Parameters

- **RA**: Specifies target general-purpose register where result of operation is stored.

See [Extended Mnemonics of Fixed-Point Rotate and Shift Instructions](#) for more information.
RS Specifies source general-purpose register for operation.
SH Specifies shift value for operation.
MB Specifies begin value of mask for operation.
ME Specifies end value of mask for operation.
BM Specifies value of 32-bit mask.

Examples
1. The following code rotates the contents of GPR 4 to the left by 2 bits and logically ANDs the result with a mask of 29 ones:
   
   Assume GPR 4 contains 0x9000 3000.
   Assume GPR 6 contains 0xFFFF FFFF.
   r1wimn 6,4,2,0,0x1D
   # GPR 6 now contains 0x4000 C000.
   # Under the same conditions
   # r1wimn 6,4,2,0xFFFFFC
   # will produce the same result.

2. The following code rotates the contents of GPR 4 to the left by 2 bits, logically ANDs the result with a mask of 29 ones, and sets Condition Register Field 0 to reflect the result of the operation:
   
   Assume GPR 4 contains 0xB004 3000.
   Assume GPR 6 contains 0xFFFF FFFF.
   r1wimn 6,4,2,0,0x1D
   # GPR 6 now contains 0xC010 C000.
   # CRF 0 now contains 0x8.
   # Under the same conditions
   # r1wimn 6,4,2,0xFFFFFC
   # will produce the same result.

Related Information
- Fixed-Point Processor
- Fixed-Point Rotate and Shift Instructions

rlwnm or rlnm (Rotate Left Word Then AND with Mask) Instruction

Purpose
Rotates the contents of a general-purpose register to the left by the number of bits specified in another general-purpose register, logically ANDs the rotated data with the generated mask, and stores the result in a third general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>23</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-25</td>
<td>MB</td>
</tr>
<tr>
<td>26-30</td>
<td>ME</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
PowerPC

\text{rlwnm} \quad RA \quad RS \quad RB \quad MB \quad ME
\text{rlwnm.} \quad RA \quad RS \quad RB \quad MB \quad ME
\text{rlwnm} \quad RA \quad RS \quad SH \quad BM
\text{rlwnm.} \quad RA \quad RS \quad SH \quad BM

POWER family

\text{rlm} \quad RA \quad RS \quad RB \quad MB \quad ME
\text{rlm.} \quad RA \quad RS \quad RB \quad MB \quad ME
\text{rlm} \quad RA \quad RS \quad SH \quad BM
\text{rlm.} \quad RA \quad RS \quad SH \quad BM

See Extended Mnemonics of Fixed-Point Rotate and Shift Instructions for more information.

Description

The \text{rlwnm} and \text{rlm} instructions rotate the contents of the source general-purpose register (GPR) \text{RS} to the left by the number of bits specified by bits 27-31 of GPR \text{RB}, logically AND the rotated data with a 32-bit generated mask defined by the values in Mask Begin (\text{MB}) and Mask End (\text{ME}), and store the result in GPR \text{RA}.

Consider the following when using the \text{rlwnm} and \text{rlm} instructions:

\begin{itemize}
  \item If the \text{MB} value is less than the \text{ME} value + 1, then the mask bits between and including the starting point and the end point are set to ones. All other bits are set to zeros.
  \item If the \text{MB} value is the same as the \text{ME} value + 1, then all 32 mask bits are set to ones.
  \item If the \text{MB} value is greater than the \text{ME} value + 1, then all of the mask bits between and including the \text{ME} value +1 and the \text{MB} value - 1 are set to zeros. All other bits are set to ones.
\end{itemize}

The \text{BM} parameter can also be used to specify the mask for these instructions. The assembler will generate the \text{MB} and \text{ME} parameters from the \text{BM} value.

The \text{rlwnm} and \text{rlm} instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rlwnm</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>rlwnm.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>rlm</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>rlm.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The syntax forms of the \text{rlwnm} and \text{rlm} instructions never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instructions affect the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

\begin{itemize}
  \item \text{RA} \quad \text{Specifies target general-purpose register where result of operation is stored.}
  \item \text{RS} \quad \text{Specifies source general-purpose register for operation.}
  \item \text{RB} \quad \text{Specifies general-purpose register that contains number of bits for rotation of data.}
  \item \text{MB} \quad \text{Specifies begin value of mask for operation.}
  \item \text{ME} \quad \text{Specifies end value of mask for operation.}
\end{itemize}
SH  Specifies shift value for operation.
BM  Specifies value of 32-bit mask.

Examples
1. The following code rotates the contents of GPR 4 to the left by 2 bits, logically ANDs the result with a mask of 29 ones, and stores the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 5 contains 0x0000 0002.
# Assume GPR 6 contains 0xFFFF FFFF.
rlwnm 6,4,5,0,0x1D
# GPR 6 now contains 0xC000 C000.
# Under the same conditions
# rlwnm 6,4,5,0xFFFFFFFF
# will produce the same result.
```

2. The following code rotates GPR 4 to the left by 2 bits, logically ANDs the result with a mask of 29 ones, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 5 contains 0x0000 0002.
# Assume GPR 6 contains 0xFFFF FFFF.
rlwnm. 6,4,5,0,0x1D
# GPR 6 now contains 0xC010 C000.
# CRF 0 now contains 0x8.
# Under the same conditions
# rlwnm. 6,4,5,0xFFFFFFFF
# will produce the same result.
```

Related Information
Fixed-Point Processor.
Fixed-Point Rotate and Shift Instructions.

**rrib (Rotate Right and Insert Bit) Instruction**

**Purpose**
Rotates bit 0 in a general-purpose register to the right by the number of bits specified by another general-purpose register and stores the rotated bit in a third general-purpose register.

Note: The rrrib instruction is supported only in the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>537</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
**Description**

The rrib instruction rotates bit 0 of the source general-purpose register (GPR) RS to the right by the number of bits specified by bits 27-31 of GPR RB and then stores the rotated bit in GPR RA.

The rrib instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrib</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>rrib.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the rrib instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- **RA** Specifies target general-purpose register where result of operation is stored.
- **RS** Specifies source general-purpose register for operation.
- **RB** Specifies general-purpose register that contains the number of bits for rotation of data.

**Examples**

1. The following code rotates bit 0 of GPR 5 to the right by 4 bits and stores its value in GPR 4:

   ```
   # Assume GPR 5 contains 0x0000 0000.
   # Assume GPR 6 contains 0x0000 0004.
   # Assume GPR 4 contains 0xFFFF FFFF.
   rrib 4,5,6
   # GPR 4 now contains 0xF7FF FFFF.
   ```

2. The following code rotates bit 0 of GPR 5 to the right by 4 bits, stores its value in GPR 4, and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 5 contains 0x0004 3000.
   # Assume GPR 6 contains 0x0000 0004.
   # Assume GPR 4 contains 0x0000 0000.
   rrib. 4,5,6
   # GPR 4 now contains 0x0800 0000.
   ```

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Rotate and Shift Instructions](#)
sc (System Call) Instruction

Purpose
Calls the system to provide a service.

Note: The sc instruction is supported only in the PowerPC architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>17</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-19</td>
<td>///</td>
</tr>
<tr>
<td>20-26</td>
<td>LEV</td>
</tr>
<tr>
<td>27-29</td>
<td>///</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

PowerPC

sc "LEV"

Description
The sc instruction causes a system call interrupt. The effective address (EA) of the instruction following the sc instruction is placed into the Save Restore Register 0 (SRR0). Bits 0, 5-9, and 16-31 of the Machine State Register (MSR) are placed into the corresponding bits of Save Restore Register 1 (SRR1). Bits 1-4 and 10-15 of SRR1 are set to undefined values.

The sc instruction serves as both a basic and an extended mnemonic. In the extended form, the LEV field is omitted and assumed to be 0.

The sc instruction has one syntax form. The syntax form does not affect the Machine State Register.

Note: The sc instruction has the same op code as the "svc (Supervisor Call) Instruction" on page 446.

Parameters

LEV Must be 0 or 1.

Related Information

"svc (Supervisor Call) Instruction" on page 446.

"Branch Processor" on page 19

"System Call Instruction" on page 20

"Functional Differences for POWER family and PowerPC Instructions" on page 114
scv (System Call Vectored) Instruction

Purpose
Calls the system to provide a service.

Note: The scv instruction is supported only in the PowerPC architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>17</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-19</td>
<td>///</td>
</tr>
<tr>
<td>20-26</td>
<td>LEV</td>
</tr>
<tr>
<td>27-29</td>
<td>///</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
</tr>
</tbody>
</table>

PowerPC

```
scv  "LEV"
```

Description
The scv instruction causes a system call interrupt. The effective address (EA) of the instruction following the scv instruction is placed into the Link Register. Bits 0-32, 37-41, and 48-63 of the Machine State Register (MSR) are placed into the corresponding bits of Count Register. Bits 33-36 and 42-47 of the Count Register are set to undefined values.

The scv instruction has one syntax form. The syntax form does not affect the Machine State Register.

Note: The scv instruction has the same op code as the "svc (Supervisor Call) Instruction" on page 446.

Parameters

LEV Must be 0 or 1.

Related Information
"svc (Supervisor Call) Instruction" on page 446.

"Branch Processor" on page 19.

"System Call Instruction" on page 20.

"Functional Differences for POWER family and PowerPC Instructions" on page 114.
si (Subtract Immediate) Instruction

**Purpose**
Subtracts the value of a signed integer from the contents of a general-purpose register and places the result in a general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>12</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>SI</td>
</tr>
</tbody>
</table>

si **RT RA SINT**

**Description**
The si instruction subtracts the 16-bit signed integer specified by the SINT parameter from the contents of general-purpose register (GPR) RA and stores the result in the target GPR RT. This instruction has the same effect as the ai instruction used with a negative SINT value. The assembler negates SINT and places this value (SI) in the machine instruction:

ai RT, RA, -SINT

The si instruction has one syntax form and can set the Carry Bit of the Fixed-Point Exception Register; it never affects Condition Register Field 0.

**Parameters**
RT Specifies target general-purpose register for operation.
RA Specifies source general-purpose register for operation.
SINT Specifies 16-bit signed integer for operation.
SI Specifies the negative of the SINT value.

**Examples**
The following code subtracts 0xFFFF F800 from the contents of GPR 4, stores the result in GPR 6, and sets the Carry bit in the Fixed-Point Exception Register to reflect the result of the operation:

```plaintext
# Assume GPR 4 contains 0x0000 0000
si 6, 4, 0xFFFFF800
# GPR 6 now contains 0x0000 0800
# This instruction has the same effect as
# ai 6, 4, -0xFFFFF800.
```

**Related Information**
The addic or ai (Add Immediate Carrying) instruction.
si. (Subtract Immediate and Record) Instruction

Purpose
Subtracts the value of a signed integer from the contents of a general-purpose register and places the result in a second general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>13</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>SI</td>
</tr>
</tbody>
</table>

si.  \[RT, RA, SINT\]

Description
The si. instruction subtracts the 16-bit signed integer specified by the SINT parameter from the contents of general-purpose register (GPR) RA and stores the result into the target GPR RT. This instruction has the same effect as the ai. instruction used with a negative SINT. The assembler negates SINT and places this value (SI) in the machine instruction:

ai. RT, RA, -SINT

The si. instruction has one syntax form and can set the Carry Bit of the Fixed-Point Exception Register. This instruction also affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, or Summary Overflow (SO) bit in Condition Register Field 0.

Parameters

RT  Specifies target general-purpose register for operation.
RA  Specifies source general-purpose register for operation.
SINT Specifies 16-bit signed integer for operation.
SI  Specifies the negative of the SINT value.

Examples
The following code subtracts 0xFFFF F800 from the contents of GPR 4, stores the result in GPR 6, and sets the Carry bit in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```c
# Assume GPR 4 contains 0xEFFF FFFF.
si. 6,4,0xFFFFF800
# GPR 6 now contains 0xF000 07FF.
# This instruction has the same effect as
# ai. 6,4,-0xFFFFF800.
```

Related Information
The addic. or ai. (Add Immediate Carrying and Record) instruction.
sld (Shift Left Double Word) Instruction

**Purpose**
Shift the contents of a general purpose register left by the number of bits specified by the contents of another general purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-30</td>
<td>27</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family

sld
\[ RA, RS, RB (OE=0 Rc=0) \]
sld.
\[ RA, RS, RB (OE=0 Rc=1) \]

**Description**
The contents of general purpose register (GPR) \( RS \) are shifted left the number of bits specified by the low-order seven bits of GPR \( RB \). Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The result is placed into GPR \( RA \). Shift amounts from 64 to 127 give a zero result.

Other registers altered:
- Condition Register (CR0 field):
  - Affected: LT, GT, EQ, SO (if \( Rc = 1 \))

**Parameters**
- \( RA \) Specifies target general-purpose register for the result of the operation.
- \( RS \) Specifies source general-purpose register containing the operand for the shift operation.
- \( RB \) The low-order seven bits specify the distance to shift the operand.

**Implementation**
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

sle (Shift Left Extended) Instruction

**Purpose**
Shifts the contents of a general-purpose register to the left by a specified number of bits, puts a copy of the rotated data in the MQ Register, and places the result in another general-purpose register.

**Note:** The sle instruction is supported only in the POWER family architecture.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>153</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family
sle
sle.

Description

The `sle` instruction rotates the contents of the source general-purpose register (GPR) `RS` to the left by `N` bits, where `N` is the shift amount specified in bits 27-31 of GPR `RB`. The instruction also stores the rotated word in the MQ Register and the logical AND of the rotated word and the generated mask in GPR `RA`. The mask consists of 32 minus `N` ones followed by `N` zeros.

The `sle` instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sle</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>sle.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the `sle` instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

- `RA` Specifies target general-purpose register where result of operation is stored.
- `RS` Specifies source general-purpose register for operation.
- `RB` Specifies source general-purpose register for operation.

Examples

1. The following code rotates the contents of GPR 4 to the left by 4 bits, places a copy of the rotated data in the MQ Register, and places the result of ANDing the rotated data with a mask into GPR 6:

   ```
   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 5 contains 0x0000 0004.
   sle 6,4,5
   # GPR 6 now contains 0x0003 0000.
   # The MQ Register now contains 0x0003 0009.
   ```

2. The following code rotates the contents of GPR 4 to the left by 4 bits, places a copy of the rotated data in the MQ Register, places the result of ANDing the rotated data with a mask into GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:
# Assume GPR 4 contains 0x0004 3000.
# Assume GPR 5 contains 0x0000 0004.
sle. 6,4,5
# GPR 6 now contains 0x0043 0000.
# The MQ Register now contains 0x0043 0008.
# Condition Register Field 0 now contains 0x4.

Related Information
- Fixed-Point Processor
- Fixed-Point Rotate and Shift Instructions

**sleq (Shift Left Extended with MQ) Instruction**

**Purpose**
Rotates the contents of a general-purpose register to the left by a specified number of bits, merges the result with the contents of the MQ Register under control of a mask, and places the rotated word in the MQ Register and the masked result in another general-purpose register.

**Note:** The `sleq` instruction is supported only in the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>217</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**Description**
The `sleq` instruction rotates the contents of the source general-purpose register (GPR) `RS` left `N` bits, where `N` is the shift amount specified in bits 27-31 of GPR `RB`. The instruction merges the rotated word with the contents of the MQ Register under control of a mask, and stores the rotated word in the MQ Register and merged word in GPR `RA`. The mask consists of 32 minus `N` ones followed by `N` zeros.

The `sleq` instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sleq</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>sleq.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>
The two syntax forms of the `sleq` instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- **RA** Specifies target general-purpose register where result of operation is stored.
- **RS** Specifies source general-purpose register for operation.
- **RB** Specifies source general-purpose register for operation.

**Examples**

1. The following code rotates the contents of GPR 4 to the left by 4 bits, merges the rotated data with the contents of the MQ Register under a generated mask, and places the rotated word in the MQ Register and the result in GPR 6:

   ```
   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 5 contains 0x0000 0004.
   # Assume the MQ Register contains 0xFFFF FFFF.
   sleq 6,4,5
   # GPR 6 now contains 0x0003 000F.
   # The MQ Register now contains 0x0003 0009.
   ```

2. The following code rotates the contents of GPR 4 to the left by 4 bits, merges the rotated data with the contents of the MQ Register under a generated mask, places the rotated word in the MQ Register and the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0xB004 3000.
   # Assume GPR 5 contains 0x0000 0004.
   # Assume the MQ Register contains 0xFFFF FFFF.
   sleq 6,4,5
   # GPR 6 now contains 0x0043 000F.
   # The MQ Register now contains 0x0043 000B.
   # Condition Register Field 0 now contains 0x4.
   ```

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Rotate and Shift Instructions](#)

**sliq (Shift Left Immediate with MQ) Instruction**

**Purpose**

Shifts the contents of a general-purpose register to the left by a specified number of bits in an immediate value, and places the rotated contents in the MQ Register and the result in another general-purpose register.

**Note:** The `sliq` instruction is supported only in the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>SH</td>
</tr>
</tbody>
</table>
### Description

The **sliq** instruction rotates the contents of the source general-purpose register (GPR) $RS$ to the left by $N$ bits, where $N$ is the shift amount specified by $SH$. The instruction stores the rotated word in the MQ Register and the logical AND of the rotated word and places the generated mask in GPR $RA$. The mask consists of $32$ minus $N$ ones followed by $N$ zeros.

The **sliq** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field $0$.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field $0$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sliq</strong></td>
<td>None</td>
<td>None</td>
<td>$0$</td>
<td>None</td>
</tr>
<tr>
<td><strong>sliq.</strong></td>
<td>None</td>
<td>None</td>
<td>$1$</td>
<td>LT, GT, EQ, SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the **sliq** instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to $1$, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field $0$.

### Parameters

- **$RA$**: Specifies target general-purpose register where result of operation is stored.
- **$RS$**: Specifies source general-purpose register for operation.
- **$SH$**: Specifies immediate value for shift amount.

### Examples

1. The following code rotates the contents of GPR 4 to the left by 20 bits, ANDs the rotated data with a generated mask, and places the rotated word into the MQ Register and the result in GPR 6:

   ```
   # Assume GPR 4 contains 0x1234 5678.
   sliq 6,4,0x14  
   # GPR 6 now contains 0x6780 0000.
   # MQ Register now contains 0x6781 2345.
   ```

2. The following code rotates the contents of GPR 4 to the left by 16 bits, ANDs the rotated data with a generated mask, places the rotated word into the MQ Register and the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x1234 5678.
   sliq. 6,4,0x10  
   # GPR 6 now contains 0x5678 0000.
   # The MQ Register now contains 0x5678 1234.
   # Condition Register Field 0 now contains 0x4.
   ```
slliq (Shift Left Long Immediate with MQ) Instruction

Purpose
Rotates the contents of a general-purpose register to the left by a specified number of bits in an immediate value, merges the result with the contents of the MQ Register under control of a mask, and places the rotated word in the MQ Register and the masked result in another general-purpose register.

Note: The slliq instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>SH</td>
</tr>
<tr>
<td>21-30</td>
<td>248</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family
slliq, slliq.

Description
The slliq instruction rotates the contents of the source general-purpose register (GPR) $RS$ to the left by $N$ bits, where $N$ is the shift amount specified in $SH$, merges the result with the contents of the MQ Register, and stores the rotated word in the MQ Register and the final result in GPR $RA$. The mask consists of 32 minus $N$ ones followed by $N$ zeros.

The slliq instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>slliq</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>slliq.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the slliq instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

$RA$ Specifies target general-purpose register where result of operation is stored.
**Examples**

1. The following code rotates the contents of GPR 4 to the left by 3 bits, merges the rotated data with the contents of the MQ Register under a generated mask, and places the rotated word in the MQ Register and the result in GPR 6:

```assembly
# Assume GPR 4 contains 0x9000 3000.
# Assume the MQ Register contains 0xFFFF FFFF.
sllq 6,4,0x3
# GPR 6 now contains 0x8001 8007.
# The MQ Register now contains 0x8001 8004.
```

2. The following code rotates the contents of GPR 4 to the left by 4 bits, merges the rotated data with the contents of the MQ Register under a generated mask, places the rotated word in the MQ Register and the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

```assembly
# Assume GPR 4 contains 0xB004 3000.
# Assume the MQ Register contains 0xFFFF FFFF.
sllq 6,4,0x4
# GPR 6 now contains 0x0043 000F.
# The MQ Register contains 0x0043 000B.
# Condition Register Field 0 now contains 0x4.
```

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Rotate and Shift Instructions](#)

### sllq (Shift Left Long with MQ) Instruction

**Purpose**

Rotates the contents of a general-purpose register to the left by the number of bits specified in a general-purpose register, merges either the rotated data or a word of zeros with the contents of the MQ Register, and places the result in a third general-purpose register.

**Note:** The *sliq* instruction is supported only in the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>216</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**POWER family**

- `sllq`  
  - ![RA]  
  - ![RS]  
  - ![RB]  

- `sllq`  
  - ![RA]  
  - ![RS]  
  - ![RB]
## Description

The `sllq` instruction rotates the contents of the source general-purpose register (GPR) `RS` to the left by `N` bits, where `N` is the shift amount specified in bits 27-31 of GPR `RB`. The merge depends on the value of bit 26 in GPR `RB`.

Consider the following when using the `sllq` instruction:

- If bit 26 of GPR `RB` is 0, then a mask of `N` zeros followed by 32 minus `N` ones is generated. The rotated word is then merged with the contents of the MQ Register under the control of this generated mask.
- If bit 26 of GPR `RB` is 1, then a mask of `N` ones followed by 32 minus `N` zeros is generated. A word of zeros is then merged with the contents of the MQ Register under the control of this generated mask.

The resulting merged word is stored in GPR `RA`. The MQ Register is not altered.

The `sllq` instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sllq</code></td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>sllq</code></td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the `sllq` instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

## Parameters

- **RA**: Specifies target general-purpose register where result of operation is stored.
- **RS**: Specifies source general-purpose register for operation.
- **RB**: Specifies source general-purpose register for operation.

## Examples

1. The following code rotates the contents of GPR 4 to the left by 4 bits, merges a word of zeros with the contents of the MQ Register under a mask, and places the merged result in GPR 6:

   ```
   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 5 contains 0x0000 0024.
   # Assume MQ Register contains 0xABCD EFAB.
   sllq 6,4,5
   # GPR 6 now contains 0xABCD EFA0.
   # The MQ Register remains unchanged.
   ```

2. The following code rotates the contents of GPR 4 to the left by 4 bits, merges the rotated data with the contents of the MQ Register under a mask, places the merged result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x8004 3000.
   # Assume GPR 5 contains 0x0000 0004.
   # Assume MQ Register contains 0xFFFF FFFF.
   sllq. 6,4,5
   # GPR 6 now contains 0x0043 000F.
   # The MQ Register remains unchanged.
   # Condition Register Field 0 now contains 0x4.
   ```
Related Information

Fixed-Point Processor.

Fixed-Point Rotate and Shift Instructions.

slq (Shift Left with MQ) Instruction

Purpose
Rotates the contents of a general-purpose register to the left by the number of bits specified in a general-purpose register, places the rotated word in the MQ Register, and places the logical AND of the rotated word and a generated mask in a third general-purpose register.

Note: The slq instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>152</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family

slq

slq.

Description
The slq instruction rotates the contents of the source general-purpose register (GPR) RS to the left by N bits, where N is the shift amount specified in bits 27-31 of GPR RB, and stores the rotated word in the MQ Register. The mask depends on bit 26 of GPR RB.

Consider the following when using the slq instruction:
• If bit 26 of GPR RB is 0, then a mask of 32 minus N ones followed by N zeros is generated.
• If bit 26 of GPR RB is 1, then a mask of all zeros is generated.

This instruction then stores the logical AND of the rotated word and the generated mask in GPR RA.

The slq instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>slq</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>slq.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EO,SO</td>
</tr>
</tbody>
</table>
The two syntax forms of the slq instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

RA Specifies target general-purpose register where result of operation is stored.

RS Specifies source general-purpose register for operation.

RB Specifies source general-purpose register for operation.

**Examples**

1. The following code rotates the contents of GPR 4 to the left by 4 bits, places the rotated word in the MQ Register, and places logical AND of the rotated word and the generated mask in GPR 6:

   ```
   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 5 contains 0x0000 0024.
   slq 6,4,5
   # GPR 6 now contains 0x0000 0000.
   # The MQ Register now contains 0x0003 0009.
   ``

2. The following code rotates the contents of GPR 4 to the left by 4 bits, places the rotated word in the MQ Register, places logical AND of the rotated word and the generated mask in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0xB004 3000.
   # Assume GPR 5 contains 0x0000 0004.
   slq 6,4,5
   # GPR 6 now contains 0x0043 0000.
   # The MQ Register now contains 0x0043 0008.
   # Condition Register Field 0 now contains 0x4.
   ``

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Rotate and Shift Instructions](#)

---

**slw or sl (Shift Left Word) Instruction**

**Purpose**

Rotates the contents of a general-purpose register to the left by a specified number of bits and places the masked result in another general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>24</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**PowerPC**

slw

**Chapter 8. Instruction Set** 373
**Description**

The `slw` and `sl` instructions rotate the contents of the source general-purpose register (GPR) `RS` to the left `N` bits, where `N` is the shift amount specified in bits 27-31 of GPR `RB`, and store the logical AND of the rotated word and the generated mask in GPR `RA`.

Consider the following when using the `slw` and `sl` instructions:
- If bit 26 of GPR `RB` is 0, then a mask of `32-N` ones followed by `N` zeros is generated.
- If bit 26 of GPR `RB` is 1, then a mask of all zeros is generated.

The `slw` and `sl` instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>slw</code></td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>slw</code></td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td><code>sl</code></td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>sl</code></td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the `slw` instruction, and the two syntax forms of the `sl` instruction, never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, these instructions affect the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- **RA** Specifies target general-purpose register where result of operation is stored.
- **RS** Specifies source general-purpose register for operation.
- **RB** Specifies source general-purpose register for operation.

**Examples**

1. The following code rotates the contents of GPR 4 to the left by 15 bits and stores the result of ANDing the rotated data with a generated mask in GPR 6:

```
/* Assume GPR 5 contains 0x0000 002F. */
/* Assume GPR 4 contains 0xFFFF FFFF. */
slw 6,4,5
/* GPR 6 now contains 0x0000 0000. */
```

2. The following code rotates the contents of GPR 4 to the left by 5 bits, stores the result of ANDing the rotated data with a generated mask in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:
Related Information

Fixed-Point Processor

Fixed-Point Rotate and Shift Instructions

srad (Shift Right Algebraic Double Word) Instruction

Purpose

Algebraically shift the contents of a general purpose register right by the number of bits specified by the contents of another general purpose register. Place the result of the operation in another general purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-30</td>
<td>794</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family

srad

R wyświetlania

srad.

Description

The contents of general purpose register (GPR) RS are shifted right the number of bits specified by the low-order seven bits of GPR RB. Bits shifted out of position 63 are lost. Bit 0 of GPR RS is replicated to fill the vacated positions on the left. The result is placed into GRP RA. XER[CA] is set if GPR RS is negative and any 1 bits are shifted out of position 63; otherwise XER[CA] is cleared. A shift amount of zero causes GRP RA to be set equal to GPR RS, and XER[CA] to be cleared. Shift amounts from 64 to 127 give a result of 64 sign bits in GRP RA, and cause XER[CA] to receive the sign bit of GPR RS.

Note that the srad instruction, followed by addze, can by used to divide quickly by 2**n. The setting of the CA bit, by srad, is independent of mode.

Other registers altered:

- Condition Register (CR0 field):
  - Affected: LT, GT, EQ, SO (if Rc = 1)
- XER:
  - Affected: CA
Parameters

RA Specifies target general-purpose register for the result of the operation.
RS Specifies source general-purpose register containing the operand for the shift operation.
RB Specifies the distance to shift the operand.

Implementation

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

sradi (Shift Right Algebraic Double Word Immediate) Instruction

Purpose

Algebraically shift the contents of a general purpose register right by the number of bits specified by the immediate value. Place the result of the operation in another general purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>sh</td>
</tr>
<tr>
<td>21-29</td>
<td>413</td>
</tr>
<tr>
<td>30</td>
<td>sh</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family

sradi | RA | RS | SH | (Rc=0)
sradi | RA | RS | SH | (Rc=1)

Description

The contents of general purpose register (GPR) RS are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 0 of GPR RS is replicated to fill the vacated positions on the left. The result is placed into GPR RA. XER[CA] is set if GPR RS is negative and any 1 bits are shifted out of position 63; otherwise XER[CA] is cleared. A shift amount of zero causes GPR RA to be set equal to GPR RS, and XER[CA] to be cleared.

Note that the sradi instruction, followed by addze, can by used to divide quickly by $2^n$. The setting of the CA bit, by sradi, is independent of mode.

Other registers altered:

• Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
• XER:
  Affected: CA
Parameters

RA Specifies target general-purpose register for the result of the operation.
RS Specifies source general-purpose register containing the operand for the shift operation.
SH Specifies shift value for operation.

Implementation

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

sraiq (Shift Right Algebraic Immediate with MQ) Instruction

Purpose

Rotates the contents of a general-purpose register to the left by a specified number of bits, merges the rotated data with a word of 32 sign bits from that general-purpose register under control of a generated mask, and places the rotated word in the MQ Register and the merged result in another general-purpose register.

Note: The sraiq instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>SH</td>
</tr>
<tr>
<td>21-30</td>
<td>952</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family

sraiq
sraiq.

Description

The sraiq instruction rotates the contents of the source general-purpose register (GPR) RS to the left by 32 minus N bits, where N is the shift amount specified by SH, merges the rotated data with a word of 32 sign bits from GPR RS under control of a generated mask, and stores the rotated word in the MQ Register and the merged result in GPR RA. A word of 32 sign bits is generated by taking the sign bit of a GPR and repeating it 32 times to make a full word. This word can be either 0x0000 0000 or 0xFFFF FFFF depending on the value of the GPR. The mask consists of N zeros followed by 32 minus N ones.

This instruction then ANDs the rotated data with the complement of the generated mask, ORs the 32-bit result together, and ANDs the bit result with bit 0 of GPR RS to produce the Carry bit (CA).

The sraiq instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.
The two syntax forms of the `sraiq` instruction always affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

### Parameters

- **RA** Specifies target general-purpose register where result of operation is stored.
- **RS** Specifies source general-purpose register for operation.
- **SH** Specifies immediate value for shift amount.

### Examples

1. The following code rotates the contents of GPR 4 to the left by 28 bits, merges the result with 32 sign bits under control of a generated mask, stores the result in GPR 6, and sets the Carry bit in the Fixed-Point Exception Register to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x9000 3000.
   sraiq 6,4,0x4
   # GPR 6 now contains 0xF900 0300.
   # MQ now contains 0x0900 0300.
   ```

2. The following code rotates the contents of GPR 4 to the left by 28 bits, merges the result with 32 sign bits under control of a generated mask, stores the result in GPR 6, and sets the Carry bit in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0xB004 3000.
   sraiq 6,4,0x4
   # GPR 6 now contains 0xFB00 4300.
   # MQ now contains 0x0B00 4300.
   # Condition Register Field 0 now contains 0x8.
   ```

### Related Information

The [addze](#) or [aze](#) (Add to Zero Extended) instruction.

[Fixed-Point Processor](#)

[Fixed-Point Rotate and Shift Instructions](#)

### sraq (Shift Right Algebraic with MQ) Instruction

#### Purpose

Rotates a general-purpose register a specified number of bits to the left, merges the result with a word of 32 sign bits from that general-purpose register under control of a generated mask, and places the rotated word in the MQ Register and the merged result in another general-purpose register.

**Note:** The `sraq` instruction is supported only in the POWER family architecture.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>920</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family
sraq
sraq.

Description
The sraq instruction rotates the contents of the source general-purpose register (GPR) RS to the left by 32 minus N bits, where N is the shift amount specified in bits 27-31 of GPR RB. The instruction then merges the rotated data with a word of 32 sign bits from GPR RS under control of a generated mask and stores the merged word in GPR RA. The rotated word is stored in the MQ Register. The mask depends on the value of bit 26 in GPR RB.

Consider the following when using the sraq instruction:
- If bit 26 of GPR RB is 0, then a mask of N zeros followed by 32 minus N ones is generated.
- If bit 26 of GPR RB is 1, then a mask of all zeros is generated.

A word of 32 sign bits is generated by taking the sign bit of a GPR and repeating it 32 times to make a full word. This word can be either 0x0000 0000 or 0xFFFF FFFF depending on the value of the GPR.

This instruction then ANDs the rotated data with the complement of the generated mask, ORs the 32-bit result together, and ANDs the bit result with bit 0 of GPR RS to produce the Carry bit (CA).

The sraq instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sraq</td>
<td>None</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>sraq.</td>
<td>None</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the sraq instruction always affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters
- RA Specifies target general-purpose register where result of operation is stored.
- RS Specifies source general-purpose register for operation.
- RB Specifies source general-purpose register for operation.
Examples
1. The following code rotates the contents of GPR 4 to the left by 28 bits, merges the result with 32 sign
bits under control of a generated mask, places the result in GPR 6 and the rotated word in the MQ
Register, and sets the Carry bit in the Fixed-Point Exception Register to reflect the result of the
operation:
   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 7 contains 0x0000 0024.
   sraq 6,4,7
   # GPR 6 now contains 0xFFFF FFFF.
   # The MQ Register now contains 0x0900 0300.

2. The following code rotates the contents of GPR 4 to the left by 28 bits, merges the result with 32 sign
bits under control of a generated mask, places the result in GPR 6 and the rotated word in the MQ
Register, and sets the Carry bit in the Fixed-Point Exception Register and Condition Register Field 0 to
reflect the result of the operation:
   # Assume GPR 4 contains 0x8004 3000.
   # Assume GPR 7 contains 0x0000 0004.
   sraq 6,4,7
   # GPR 6 now contains 0xFB00 4300.
   # The MQ Register now contains 0x0800 4300.
   # Condition Register Field 0 now contains 0x4.

Related Information
- Fixed-Point Processor
- Fixed-Point Rotate and Shift Instructions

sraw or sra (Shift Right Algebraic Word) Instruction

Purpose
Rotates the contents of a general-purpose register to the left by a specified number of bits, merges the
rotated data with a word of 32 sign bits from that register under control of a generated mask, and places
the result in another general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>792</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC
- sraw  
  RA | RS | RB
- sraw.

POWER family
- sra  
  RA | RS | RB
- sra.
Description
The `sraw` and `sra` instructions rotate the contents of the source general-purpose register (GPR) `RS` to the left by 32 minus `N` bits, where `N` is the shift amount specified in bits 27-31 of GPR `RB`, and merge the rotated word with a word of 32 sign bits from GPR `RS` under control of a generated mask. A word of 32 sign bits is generated by taking the sign bit of a GPR and repeating it 32 times to make a full word. This word can be either 0x0000 0000 or 0xFFFF FFFF depending on the value of the GPR.

The mask depends on the value of bit 26 in GPR `RB`.

Consider the following when using the `sraw` and `sra` instructions:
- If bit 26 of GPR `RB` is zero, then a mask of `N` zeros followed by 32 minus `N` ones is generated.
- If bit 26 of GPR `RB` is one, then a mask of all zeros is generated.

The merged word is placed in GPR `RA`. The `sraw` and `sra` instructions then AND the rotated data with the complement of the generated mask, OR the 32-bit result together, and AND the bit result with bit 0 of GPR `RS` to produce the Carry bit (CA).

The `sraw` and `sra` instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sraw</code></td>
<td>None</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>sraw.</code></td>
<td>None</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td><code>sra</code></td>
<td>None</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>sra.</code></td>
<td>None</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the `sraw` instruction, and the two syntax forms of the `sra` instruction, always affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instructions affect the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters
- `RA` Specifies target general-purpose register where result of operation is stored.
- `RS` Specifies source general-purpose register for operation.
- `RB` Specifies source general-purpose register for operation.

Examples
1. The following code rotates the contents of GPR 4 to the left by 28 bits, merges the result with 32 sign bits under control of a generated mask, stores the result in GPR 6, and sets the Carry bit in the Fixed-Point Exception Register to reflect the result of the operation:
   ```
   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 5 contains 0x0000 0024.
   sraw 6,4,5
   # GPR 6 now contains 0xFFFF FFFF.
   ```
2. The following code rotates the contents of GPR 4 to the left by 28 bits, merges the result with 32 sign bits under control of a generated mask, stores the result in GPR 6, and sets the Carry bit in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:
Related Information

The `addze` or `aze` (Add to Zero Extended) instruction.

Fixed-Point Processor

Fixed-Point Rotate and Shift Instructions

---

**srawi or srai (Shift Right Algebraic Word Immediate) Instruction**

**Purpose**

Rotates the contents of a general-purpose register a specified number of bits to the left, merges the rotated data with a word of 32 sign bits from that register under control of a generated mask, and places the result in another general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>SH</td>
</tr>
<tr>
<td>21-30</td>
<td>824</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**PowerPC**

`srawi RA RS SH`

`srawi. RA RS SH`

**POWER family**

`srai RA RS SH`

`srai. RA RS SH`

**Description**

The `srawi` and `srai` instructions rotate the contents of the source general-purpose register (GPR) `RS` to the left by 32 minus `N` bits, where `N` is the shift amount specified by `SH`, merge the rotated data with a word of 32 sign bits from GPR `RS` under control of a generated mask, and store the merged result in GPR `RA`. A word of 32 sign bits is generated by taking the sign bit of a GPR and repeating it 32 times to make a full word. This word can be either `0x0000 0000` or `0xFFFF FFFF` depending on the value of the GPR. The mask consists of `N` zeros followed by 32 minus `N` ones.

The `srawi` and `srai` instructions then AND the rotated data with the complement of the generated mask, OR the 32-bit result together, and AND the bit result with bit 0 of GPR `RS` to produce the Carry bit (CA).
The `srawi` and `srai` instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>srawi</td>
<td>None</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>srawi.</td>
<td>None</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>srai</td>
<td>None</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>srai.</td>
<td>None</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the `srawi` instruction, and the two syntax forms of the `srai` instruction, always affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instructions affect the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

RA Specifications target general-purpose register where result of operation is stored.

RS Specifies source general-purpose register for operation.

SH Specifies immediate value for shift amount.

**Examples**

1. The following code rotates the contents of GPR 4 to the left by 28 bits, merges the result with 32 sign bits under control of a generated mask, stores the result in GPR 6, and sets the Carry bit in the Fixed-Point Exception Register to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x9000 3000.
   srawi 6,4,0x4
   # GPR 6 now contains 0xF900 0300.
   ```

2. The following code rotates the contents of GPR 4 to the left by 28 bits, merges the result with 32 sign bits under control of a generated mask, places the result in GPR 6, and sets the Carry bit in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0xB004 3000.
   srawi. 6,4,0x4
   # GPR 6 now contains 0xFB00 4300.
   # Condition Register Field 0 now contains 0x8.
   ```

**Related Information**

The `addze` or `aze` (Add to Zero Extended) instruction.

Fixed-Point Processor

Fixed-Point Rotate and Shift Instructions

**srd (Shift Right Double Word) Instruction**

**Purpose**

Shift the contents of a general purpose register right by the number of bits specified by the contents of another general purpose register.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-30</td>
<td>539</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family

<table>
<thead>
<tr>
<th>srd</th>
<th>RA</th>
<th>RS</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>srd.</td>
<td>RA</td>
<td>RS</td>
<td>RB</td>
</tr>
</tbody>
</table>

Description

The contents of general purpose register (GPR) RS are shifted right the number of bits specified by the low-order seven bits of GPR RB. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The result is placed into GRP RA. Shift amounts from 64 to 127 give a zero result.

Other registers altered:
- Condition Register (CR0 field):
  - Affected: LT, GT, EQ, SO (if Rc = 1)

Parameters

RA Specifies target general-purpose register for the result of the operation.
RS Specifies source general-purpose register containing the operand for the shift operation.
RB The low-order seven bits specify the distance to shift the operand.

Implementation

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

sre (Shift Right Extended) Instruction

Purpose

Shifts the contents of a general-purpose register to the right by a specified number of bits and places a copy of the rotated data in the MQ Register and the result in a general-purpose register.

Note: The sre instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>Bits</td>
<td>Value</td>
</tr>
<tr>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>665</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**Description**

The `sre` instruction rotates the contents of the source general-purpose register (GPR) `RS` to the left by 32 minus `N` bits, where `N` is the shift amount specified in bits 27-31 of GPR `RB`, and stores the rotated word in the MQ Register and the logical AND of the rotated word and a generated mask in GPR `RA`. The mask consists of `N` zeros followed by 32 minus `N` ones.

The `sre` instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sre</code></td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>sre.</code></td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT, GT, EQ, SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the `sre` instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- `RA` Specifies target general-purpose register where result of operation is stored.
- `RS` Specifies source general-purpose register for operation.
- `RB` Specifies source general-purpose register for operation.

**Examples**

1. The following code rotates the contents of GPR 4 to the left by 20 bits, places a copy of the rotated data in the MQ Register, and places the result of ANDing the rotated data with a mask into GPR 6:

   ```
   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 5 contains 0x0000 000C.
   sre 6,4,5
   # GPR 6 now contains 0x0000 0003.
   # The MQ Register now contains 0x0000 0003.
   ```

2. The following code rotates the contents of GPR 4 to the left by 17 bits, places a copy of the rotated data in the MQ Register, places the result of ANDing the rotated data with a mask into GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x0004 3000.
   # Assume GPR 5 contains 0x0000 000F.
   sre 6,4,5
   # GPR 6 now contains 0x0000 0003.
   # The MQ Register now contains 0x0000 0003.
   # Condition Register Field 0 now contains 0x4.
   ```
srea (Shift Right Extended Algebraic) Instruction

Purpose
Rotates the contents of a general-purpose register to the left by a specified number of bits, places a copy of the rotated data in the MQ Register, merges the rotated word and a word of 32 sign bits from the general-purpose register under control of a mask, and places the result in another general-purpose register.

Note: The srea instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>921</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

Description
The srea instruction rotates the contents of the source general-purpose register (GPR) RS to the left by 32 minus N bits, where N is the shift amount specified in bits 27-31 of GPR RB, stores the rotated word in the MQ Register, and merges the rotated word and a word of 32 sign bits from GPR RS under control of a generated mask. A word of 32 sign bits is generated by taking the sign bit of a general-purpose register and repeating it 32 times to make a full word. This word can be either 0x0000 0000 or 0xFFFF FFFF depending on the value of the general-purpose register. The mask consists of N zeros followed by 32 minus N ones. The merged word is stored in GPR RA.

This instruction then ANDs the rotated data with the complement of the generated mask, ORs together the 32-bit result, and ANDs the bit result with bit 0 of GPR RS to produce the Carry bit (CA).

The srea instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>srea</td>
<td>None</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>srea</td>
<td>None</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>
The two syntax forms of the `srea` instruction always affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- **RA**: Specifies target general-purpose register where result of operation is stored.
- **RS**: Specifies source general-purpose register for operation.
- **RB**: Specifies source general-purpose register for operation.

**Examples**

1. The following code rotates the contents of GPR 4 to the left by 28 bits, merges the result with 32 sign bits under control of a generated mask, places the rotated word in the MQ Register and the result in GPR 6, and sets the Carry bit in the Fixed-Point Exception Register to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 7 contains 0x0000 0004.
   srea 6,4,7
   # GPR 6 now contains 0xF900 0300.
   # The MQ Register now contains 0x0900 0300.
   ```

2. The following code rotates the contents of GPR 4 to the left by 28 bits, merges the result with 32 sign bits under control of a generated mask, places the rotated word in the MQ Register and the result in GPR 6, and sets the Carry bit in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0xB004 3000.
   # Assume GPR 7 contains 0x0000 0004.
   srea 6,4,7
   # GPR 6 now contains 0xFB00 4300.
   # The MQ Register now contains 0x0B00 4300.
   # Condition Register Field 0 now contains 0x8.
   ```

**Related Information**

The [addze](#) or [aze](#) (Add to Zero Extended) instruction.

- Fixed-Point Processor.
- Fixed-Point Rotate and Shift Instructions.

---

**sreq (Shift Right Extended with MQ) Instruction**

**Purpose**

Rotates the contents of a general-purpose register to the left by a specified number of bits, merges the result with the contents of the MQ Register under control of a generated mask, and places the rotated word in the MQ Register and the merged result in another general-purpose register.

**Note:** The `sreq` instruction is supported only in the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
</tbody>
</table>
### Description

The `sreq` instruction rotates the contents of the source general-purpose register (GPR) `RS` to the left by 32 minus `N` bits, where `N` is the shift amount specified in bits 27-31 of GPR `RB`, merges the rotated word with the contents of the MQ Register under a generated mask, and stores the rotated word in the MQ Register and the merged word in GPR `RA`. The mask consists of `N` zeros followed by 32 minus `N` ones.

The `sreq` instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sreq</code></td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>sreq.</code></td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT, GT, EQ, SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the `sreq` instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

### Parameters

- **RA**: Specifies target general-purpose register where result of operation is stored.
- **RS**: Specifies source general-purpose register for operation.
- **RB**: Specifies source general-purpose register for operation.

### Examples

1. The following code rotates the contents of GPR 4 to the left by 28 bits, merges the rotated data with the contents of the MQ Register under a generated mask, and places the rotated word in the MQ Register and the result in GPR 6:

   ```
   # Assume GPR 4 contains 0x9000 30F.
   # Assume GPR 7 contains 0x0000 0004.
   # Assume the MQ Register contains 0xEFFF FFFF.
   sreq 6,4,7
   # GPR 6 now contains 0xE900 0300.
   # The MQ Register now contains 0xF900 0300.
   ```

2. The following code rotates the contents of GPR 4 to the left by 28 bits, merges the rotated data with the contents of the MQ Register under a generated mask, places the rotated word in the MQ Register and the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0xB000 30F.
   # Assume GPR 18 contains 0x0000 0004.
   # Assume the MQ Register contains 0xEFFF FFFF
   ```
sreq. 6,4,18
# GPR 6 now contains 0xEB00 0300.
# The MQ Register now contains 0xFB00 0300.
# Condition Register Field 0 now contains 0x8.

Related Information
Fixed-Point Processor.

Fixed-Point Rotate and Shift Instructions.

sriq (Shift Right Immediate with MQ) Instruction

Purpose
Shifts the contents of a general-purpose register to the right by a specified number of bits and places the rotated contents in the MQ Register and the result in another general-purpose register.

Note: The sriq instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>SH</td>
</tr>
<tr>
<td>21-30</td>
<td>696</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family
sriq
sriq.

Description
The sriq instruction rotates the contents of the source general-purpose register (GPR) RS to the left 32 minus \( N \) bits, where \( N \) is the shift amount specified by SH, and stores the rotated word in the MQ Register, and the logical AND of the rotated word and the generated mask in GPR RA. The mask consists of \( N \) zeros followed by 32 minus \( N \) ones.

The sriq instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sriq</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>sriq.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the sriq instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.
Parameters

RA  Specifies target general-purpose register where result of operation is stored.
RS  Specifies source general-purpose register for operation.
SH  Specifies value for shift amount.

Examples

1. The following code rotates the contents of GPR 4 to the left by 20 bits, ANDs the rotated data with a generated mask, and places the rotated word into the MQ Register and the result in GPR 6:

    # Assume GPR 4 contains 0x9000 300F.
    srlq 6,4,0xC
    # GPR 6 now contains 0x0009 0003.
    # The MQ Register now contains 0x00F9 0003.

2. The following code rotates the contents of GPR 4 to the left by 12 bits, ANDs the rotated data with a generated mask, places the rotated word into the MQ Register and the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

    # Assume GPR 4 contains 0xB000 300F.
    srlq 6,4,0x14
    # GPR 6 now contains 0x0000 0B00.
    # The MQ Register now contains 0x0300 FB00.
    # Condition Register Field 0 now contains 0x4.

Related Information

Fixed-Point Processor
Fixed-Point Rotate and Shift Instructions

srlq (Shift Right Long Immediate with MQ) Instruction

Purpose

Rotates the contents of a general-purpose register to the left by a specified number of bits, merges the result with the contents of the MQ Register under control of a generated mask, and places the result in another general-purpose register.

Note: The srlq instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>SH</td>
</tr>
<tr>
<td>21-30</td>
<td>760</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family

srlq  

srlq.
Description

The `srliq` instruction rotates the contents of the source general-purpose register (GPR) `RS` to the left by 32 minus `N` bits, where `N` is the shift amount specified by `SH`, merges the result with the contents of the MQ Register under control of a generated mask, and stores the rotated word in the MQ Register and the merged result in GPR `RA`. The mask consists of `N` zeros followed by 32 minus `N` ones.

The `srliq` instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>srliq</code></td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>srliq</code></td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the `srliq` instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

- `RA` Specifies target general-purpose register where result of operation is stored.
- `RS` Specifies source general-purpose register for operation.
- `SH` Specifies value for shift amount.

Examples

1. The following code rotates the contents of GPR 4 to the left by 28 bits, merges the rotated data with the contents of the MQ Register under a generated mask, and places the rotated word in the MQ Register and the result in GPR 6:

   ```
   # Assume GPR 4 contains 0x9000 300F.
   # Assume the MQ Register contains 0x1111 1111.
   srliq 6,4,0x4
   # GPR 6 now contains 0x1900 0300.
   # The MQ Register now contains 0xF900 0300.
   ```

2. The following code rotates the contents of GPR 4 to the left by 28 bits, merges the rotated data with the contents of the MQ Register under a generated mask, places the rotated word in the MQ Register and the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0xB004 3000
   # Assume the MQ Register contains 0xFFFF FFFF.
   srliq. 6,4,0x4
   # GPR 6 now contains 0xFB00 4300.
   # The MQ Register contains 0x0B00 4300.
   # Condition Register Field 0 now contains 0x8.
   ```

Related Information

- Fixed-Point Processor
- Fixed-Point Rotate and Shift Instructions
srlq (Shift Right Long with MQ) Instruction

Purpose
Rotates the contents of a general-purpose register to the left by a specified number of bits, merges either the rotated data or a word of zeros with the contents of the MQ Register under control of a generated mask, and places the result in a general-purpose register.

Note: The srlq instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>728</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER family
srlq
srlq.

Description
The srlq instruction rotates the contents of the source general-purpose register (GPR) RS to the left 32 minus N bits, where N is the shift amount specified in bits 27-31 of GPR RB. The merge depends on the value of bit 26 in GPR RB.

Consider the following when using the srlq instruction:
- If bit 26 of GPR RB is 0, then a mask of N zeros followed by 32 minus N ones is generated. The rotated word is then merged with the contents of the MQ Register under control of this generated mask.
- If bit 26 of GPR RB is 1, then a mask of N ones followed by 32 minus N zeros is generated. A word of zeros is then merged with the contents of the MQ Register under control of this generated mask.

The merged word is stored in GPR RA. The MQ Register is not altered.

The srlq instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>srlq</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>srlq.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the srlq instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.
Parameters

\( RA \)  \hspace{1cm} \text{Specifies target general-purpose register where result of operation is stored.}
\( RS \)  \hspace{1cm} \text{Specifies source general-purpose register for operation.}
\( RB \)  \hspace{1cm} \text{Specifies source general-purpose register for operation.}

Examples

1. The following code rotates the contents of GPR 4 to the left by 28 bits, merges a word of zeros with the contents of the MQ Register under a mask, and places the merged result in GPR 6:

\[ \text{# Assume GPR 4 contains 0x9000 300F.} \]
\[ \text{# Assume GPR 8 contains 0x0000 0024.} \]
\[ \text{# Assume the MQ Register contains 0xFFFF FFFF.} \]
\[ \text{srlq 6,4,8} \]
\[ \text{# GPR 6 now contains 0x0FFF FFFF.} \]
\[ \text{# The MQ Register remains unchanged.} \]

2. The following code rotates the contents of GPR 4 to the left by 28 bits, merges the rotated data with the contents of the MQ Register under a mask, places the merged result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

\[ \text{# Assume GPR 4 contains 0xB004 3000.} \]
\[ \text{# Assume GPR 8 contains 0x0000 0004.} \]
\[ \text{# Assume the MQ Register contains 0xFFFF FFFF.} \]
\[ \text{srlq 6,4,8} \]
\[ \text{# GPR 6 now holds 0xFB00 4300.} \]
\[ \text{# The MQ Register remains unchanged.} \]
\[ \text{# Condition Register Field 0 now contains 0x8.} \]

Related Information

Fixed-Point Processor.

Fixed-Point Rotate and Shift Instructions.

srq (Shift Right with MQ) Instruction

Purpose

Rotates the contents of a general-purpose register to the left by a specified number of bits, places the rotated word in the MQ Register, and places the logical AND of the rotated word and a generated mask in a general-purpose register.

\textbf{Note:} The srq instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>664</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>
Description

The `srq` instruction rotates the contents of the source general-purpose register (GPR) `RS` to the left by 32 minus `N` bits, where `N` is the shift amount specified in bits 27-31 of GPR `RB`, and stores the rotated word in the MQ Register. The mask depends on bit 26 of GPR `RB`.

Consider the following when using the `srq` instruction:

- If bit 26 of GPR `RB` is 0, then a mask of `N` zeros followed by 32 minus `N` ones is generated.
- If bit 26 of GPR `RB` is 1, then a mask of all zeros is generated.

This instruction then stores the logical AND of the rotated word and the generated mask in GPR `RA`.

The `srq` instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>srq</code></td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td><code>srq.</code></td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>

The two syntax forms of the `srq` instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

- `RA` Specifies target general-purpose register where result of operation is stored.
- `RS` Specifies source general-purpose register for operation.
- `RB` Specifies source general-purpose register for operation.

Examples

1. The following code rotates the contents of GPR 4 to the left by 28 bits, places the rotated word in the MQ Register, and places logical AND of the rotated word and the generated mask in GPR 6:

```
# Assume GPR 4 holds 0x9000 30F.
# Assume GPR 25 holds 0x0000 00024.
srq 6,4,25
# GPR 6 now holds 0x0000 0000.
# The MQ Register now holds 0xF900 0300.
```

2. The following code rotates the contents of GPR 4 to the left by 28 bits, places the rotated word in the MQ Register, places logical AND of the rotated word and the generated mask in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 holds 0x8000 30F.
# Assume GPR 25 holds 0x0000 0004.
srq 6,4,8
# GPR 6 now holds 0x8000 0000.
# The MQ Register now holds 0xF800 0300.
# Condition Register Field 0 now contains 0x4.
```
Related Information
- Fixed-Point Processor
- Fixed-Point Rotate and Shift Instructions

srw or sr (Shift Right Word) Instruction

Purpose
Rotates the contents of a general-purpose register to the left by a specified number of bits and places the masked result in a general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>536</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC
- srw: RA RS RB
- srw.

POWER family
- sr: RA RS RB
- sr.

Description
The srw and sr instructions rotate the contents of the source general-purpose register (GPR) RS to the left by 32 minus \( N \) bits, where \( N \) is the shift amount specified in bits 27-31 of GPR RB, and store the logical AND of the rotated word and the generated mask in GPR RA.

Consider the following when using the srw and sr instructions:
- If bit 26 of GPR RB is 0, then a mask of \( N \) zeros followed by 32 - \( N \) ones is generated.
- If bit 26 of GPR RB is 1, then a mask of all zeros is generated.

The srw and sr instruction each have two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>srw</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>srw.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>sr</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>sr.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>
The two syntax forms of the \texttt{sr} instruction, and the two syntax forms of the \texttt{srw} instruction, never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, these instructions affect the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- \texttt{RA} Specifies target general-purpose register where result of operation is stored.
- \texttt{RS} Specifies source general-purpose register for operation.
- \texttt{RB} Specifies source general-purpose register for operation.

**Examples**

1. The following code rotates the contents of GPR 4 to the left by 28 bits and stores the result of ANDing the rotated data with a generated mask in GPR 6:
   
   \begin{verbatim}
   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 5 contains 0x0000 0024.
   srw 6,4,5
   # GPR 6 now contains 0x0000 0000.
   \end{verbatim}

2. The following code rotates the contents of GPR 4 to the left by 28 bits, stores the result of ANDing the rotated data with a generated mask in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

   \begin{verbatim}
   # Assume GPR 4 contains 0xB004 3001.
   # Assume GPR 5 contains 0x0000 0004.
   srw. 6,4,5
   # GPR 6 now contains 0x0B00 4300.
   # Condition Register Field 0 now contains 0x4.
   \end{verbatim}

**Related Information**

The \texttt{addze} or \texttt{aze} (Add to Zero Extended) instruction.

- [Fixed-Point Processor](#)
- [Fixed-Point Rotate and Shift Instructions](#)

---

**stb (Store Byte) Instruction**

**Purpose**

Stores a byte of data from a general-purpose register into a specified location in memory.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>38</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

\texttt{stb RS D RA}
Description
The **stb** instruction stores bits 24-31 of general-purpose register (GPR) *RS* into a byte of storage addressed by the effective address (EA).

If GPR *RA* is not 0, the EA is the sum of the contents of GPR *RA* and *D*, a 16-bit signed two’s complement integer sign-extended to 32 bits. If GPR *RA* is 0, then the EA is *D*.

The **stb** instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

*RS* Specifies source general-purpose register of stored data.
*D* Specifies a 16-bit, signed two’s complement integer sign-extended to 32 bits for EA calculation.
*RA* Specifies source general-purpose register for EA calculation.

Examples
The following code stores bits 24-31 of GPR 6 into a location in memory:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 4 contains address of csect data[rw].
# Assume GPR 6 contains 0x0000 0060.
.csect text[pr]
stb 6,buffer(4)
# 0x60 is now stored at the address of buffer.
```

Related Information

[Fixed-Point Processor](#).

[Fixed-Point Load and Store Instructions](#).

---

**stbu** (Store Byte with Update) Instruction

Purpose
Stores a byte of data from a general-purpose register into a specified location in memory and possibly places the address in another general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>39</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

```
stbu RS D RA
```

Description
The **stbu** instruction stores bits 24-31 of the source general-purpose register (GPR) *RS* into the byte in storage addressed by the effective address (EA).
If GPR $RA$ is not 0, the EA is the sum of the contents of GPR $RA$ and $D$, a 16-bit signed two’s complement integer sign-extended to 32 bits. If GPR $RA$ is 0, then the EA is $D$.

If $RA$ does not equal 0 and the storage access does not cause an Alignment Interrupt, then the EA is stored in GPR $RA$.

The stbu instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Parameters**

- **RS**: Specifies source general-purpose register of stored data.
- **D**: Specifies a 16-bit, signed two’s complement integer sign-extended to 32 bits for EA calculation.
- **RA**: Specifies source general-purpose register for EA calculation and possible address update.

**Examples**

The following code stores bits 24-31 of GPR 6 into a location in memory and places the address in GPR 16:

```
.ctor data[rw]
buffer: .long 0
# Assume GPR 6 contains 0x0000 0060.
# Assume GPR 16 contains the address of csect data[rw].
.ctor text[pr]
stbu 6,buffer(16)
# GPR 16 now contains the address of buffer.
# 0x60 is stored at the address of buffer.
```

**Related Information**
- [Fixed-Point Processor](#)
- [Fixed-Point Load and Store with Update Instructions](#)

### stbux (Store Byte with Update Indexed) Instruction

**Purpose**

Stores a byte of data from a general-purpose register into a specified location in memory and possibly places the address in another general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>247</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

`stbux RS RA RB`
Description
The `stbux` instruction stores bits 24-31 of the source general-purpose register (GPR) `RS` into the byte in storage addressed by the effective address (EA).

If GPR `RA` is not 0, the EA is the sum of the contents of GPR `RA` and the contents of GPR `RB`. If `RA` is 0, then the EA is the contents of GPR `RB`.

If GPR `RA` does not equal 0 and the storage access does not cause an Alignment Interrupt, then the EA is stored in GPR `RA`.

The `stbux` instruction exists only in one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters
- **RS** Specifies source general-purpose register of stored data.
- **RA** Specifies source general-purpose register for EA calculation and possible address update.
- **RB** Specifies source general-purpose register for EA calculation.

Examples
The following code stores the contents of GPR 6 into a location in memory and places the address in GPR 4:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 6 contains 0x0000 0060.
# Assume GPR 4 contains 0x0000 0000.
# Assume GPR 19 contains the address of buffer.
.csect text[pr]
stbux 6,4,19
# Buffer now contains 0x60.
# GPR 4 contains the address of buffer.
```

Related Information
- Fixed-Point Processor
- Fixed-Point Load and Store with Update Instructions

**stbx (Store Byte Indexed) Instruction**

**Purpose**
Stores a byte from a general-purpose register into a specified location in memory.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>215</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>
**Description**

The `stbx` instruction stores bits 24-31 from general-purpose register (GPR) `RS` into a byte of storage addressed by the effective address (EA). The contents of GPR `RS` are unchanged.

If GPR `RA` is not 0, the EA is the sum of the contents of GPR `RA` and the contents of GPR `RB`. If GPR `RA` is 0, then the EA is the contents of GPR `RB`.

The `stbx` instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Parameters**

- **RS**: Specifies source general-purpose register of stored data.
- **RA**: Specifies source general-purpose register for EA calculation.
- **RB**: Specifies source general-purpose register for EA calculation.

**Examples**

The following code stores bits 24-31 of GPR 6 into a location in memory:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 4 contains the address of buffer.
# Assume GPR 6 contains 0x4865 6C6F.
.csect text[pr]
stbx 6,0,4
# buffer now contains 0x6F.
```

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Load and Store Instructions](#)

---

**std (Store Double Word) Instruction**

**Purpose**

Store a double-word of data from a general purpose register into a specified memory location.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>62</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-29</td>
<td>ds</td>
</tr>
<tr>
<td>30-31</td>
<td>00</td>
</tr>
</tbody>
</table>

**POWER family**

- `std`
Description

The `std` instruction stores a double-word in storage from the source general-purpose register (GPR) `RS` into the specified location in memory referenced by the effective address (EA).

If GPR `RA` is not 0, the EA is the sum of the contents of GPR `RA` and `D`, a 16-bit, signed two’s complement integer, fullword-aligned, sign-extended to 64 bits. If GPR `RA` is 0, then the EA is `D`.

Parameters

- `RS` Specifies the source general-purpose register containing data.
- `D` Specifies a 16-bit, signed two’s complement integer sign-extended to 32 bits for EA calculation.
- `RA` Specifies source general-purpose register for EA calculation.

Implementation

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

stdcx. (Store Double Word Conditional Indexed) Instruction

Purpose

Conditionally store the contents of a general purpose register into a storage location, based upon an existing reservation.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-30</td>
<td>214</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
</tr>
</tbody>
</table>

POWER family

stdcx. `RS RA RS`

Description

If a reservation exists, and the memory address specified by the `stdcx` instruction is the same as that specified by the load and reserve instruction that established the reservation, the contents of `RS` are stored into the double-word in memory addressed by the effective address (EA); the reservation is cleared.

If GPR `RA` is not 0, the EA is the sum of the contents of GPR `RA` and `D`, a 16-bit, signed two’s complement integer, fullword-aligned, sign-extended to 64 bits. If GPR `RA` is 0, then the EA is `D`.

If a reservation exists, but the memory address specified by the `stdcx` instruction is not the same as that specified by the load and reserve instruction that established the reservation, the reservation is cleared, and it is undefined whether the contents of `RS` are stored into the double word in memory addressed by the EA.
If no reservation exists, the instruction completes without altering memory.

If the store is performed successfully, bits 0-2 of Condition Register Field 0 are set to 0b001, otherwise, they are set to 0b000. The SO bit of the XER is copied to to bit 4 of Condition Register Field 0.

The EA must be a multiple of eight. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined.

Note that, when used correctly, the load and reserve and store conditional instructions can provide an atomic update function for a single aligned word (load word and reserve and store word conditional) or double word (load double word and reserve and store double word conditional) of memory.

In general, correct use requires that load word and reserve be paired with store word conditional, and load double word and reserve with store double word conditional, with the same memory address specified by both instructions of the pair. The only exception is that an unpaired store word conditional or store double word conditional instruction to any (scratch) EA can be used to clear any reservation held by the processor.

A reservation is cleared if any of the following events occurs:

- The processor holding the reservation executes another load and reserve instruction; this clears the first reservation and establishes a new one.
- The processor holding the reservation executes a store conditional instruction to any address.
- Another processor executes any store instruction to the address associated with the reservation.
- Any mechanism, other than the processor holding the reservation, stores to the address associated with the reservation.

**Parameters**

<table>
<thead>
<tr>
<th>RS</th>
<th>Specifies source general-purpose register of stored data.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>Specifies source general-purpose register for EA calculation.</td>
</tr>
<tr>
<td>RB</td>
<td>Specifies source general-purpose register for EA calculation.</td>
</tr>
</tbody>
</table>

**Implementation**

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

**stdu (Store Double Word with Update) Instruction**

**Purpose**

Store a double-word of data from a general purpose register into a specified memory location. Update the address base.

This instruction should only be used on 64-bit PowerPC processors running a 64-bit application.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>62</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-29</td>
<td>ds</td>
</tr>
</tbody>
</table>
Description
The stdu instruction stores a double-word in storage from the source general-purpose register (GPR) RS into the specified location in memory referenced by the effective address (EA).

If GPR RA is not 0, the EA is the sum of the contents of GPR RA and D, a 16-bit, signed two’s complement integer, fullword-aligned, sign-extended to 64 bits. GRP RA is updated with the EA.

If GPR RA = 0, the instruction form is invalid.

Parameters

<table>
<thead>
<tr>
<th>RS</th>
<th>Specifies the source general-purpose register containing data.</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Specifies a 16-bit, signed two’s complement integer sign-extended to 32 bits for EA calculation.</td>
</tr>
<tr>
<td>RA</td>
<td>Specifies source general-purpose register for EA calculation.</td>
</tr>
</tbody>
</table>

Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Related Information
Fixed-Point Processor.

Fixed-Point Load and Store with Update Instructions.

stdux (Store Double Word with Update Indexed) Instruction

Purpose
Store a double-word of data from a general purpose register into a specified memory location. Update the address base.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-30</td>
<td>181</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>
**stdx (Store Double Word Indexed) Instruction**

**Purpose**

Store a double-word of data from a general purpose register into a specified memory location.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>S</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-30</td>
<td>149</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

**Description**

The `stdx` instruction stores a double-word in storage from the source general-purpose register (GPR) `RS` into the location in storage specified by the effective address (EA).

If GPR `RA` is not 0, the EA is the sum of the contents of GPR `RA` and `RB`. If GPR `RA` is 0, then the EA is `RB`. 

**Parameters**

- `RS` Specifies the source general-purpose register containing data.
- `RA` Specifies source general-purpose register for EA calculation.
- `RB` Specifies source general-purpose register for EA calculation.
Parameters

RS Specifies the source general-purpose register containing data.
RA Specifies source general-purpose register for EA calculation.
RB Specifies source general-purpose register for EA calculation.

Implementation

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

stfd (Store Floating-Point Double) Instruction

Purpose
Stores a doubleword of data in a specified location in memory.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>54</td>
</tr>
<tr>
<td>6-10</td>
<td>FRS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

stfd $FRS$, $D$( $RA$ )

Description

The stfd instruction stores the contents of floating-point register (FPR) $FRS$ into the doubleword storage addressed by the effective address (EA).

If general-purpose register (GPR) $RA$ is not 0, the EA is the sum of the contents of GPR $RA$ and $D$. The sum is a 16-bit signed two's complement integer sign-extended to 32 bits. If GPR $RA$ is 0, then the EA is $D$.

The stfd instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

$FRS$ Specifies source floating-point register of stored data.
$D$ Specifies a16-bit signed two's complement integer sign-extended to 32 bits for EA calculation.
$RA$ Specifies source general-purpose register for EA calculation.

Examples

The following code stores the contents of FPR 6 into a location in memory:

```assembly
.csect data[rw]
buffer: .long 0,0
# Assume FPR 6 contains 0x4865 6C6C 6F20 776F.
```
# Assume GPR 4 contains the address of csect data[rw].
.csect text[pr]
stfd 6, buffer(4)
# buffer now contains 0x4865 6C6C 6F20 776F.

**Related Reading**
- Floating-Point Processor
- Floating-Point Load and Store Instructions

### stfd (Store Floating-Point Double with Update) Instruction

**Purpose**
Stores a doubleword of data in a specified location in memory and in some cases places the address in a general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>55</td>
</tr>
<tr>
<td>6-10</td>
<td>FRS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

\[ \text{stfd} \quad \text{FRS, D (RA)} \]

**Description**
The \text{stfd} instruction stores the contents of floating-point register (FPR) \text{FRS} into the doubleword storage addressed by the effective address (EA).

If general-purpose register (GPR) \text{RA} is not 0, the EA is the sum of the contents of GPR \text{RA} and \text{D}. The sum is a 16-bit signed two's complement integer sign-extended to 32 bits. If GPR \text{RA} is 0, then the EA is \text{D}.

If GPR \text{RA} does not equal 0 and the storage access does not cause Alignment Interrupt or a Data Storage Interrupt, then the EA is stored in GPR \text{RA}.

The \text{stfd} instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

**Parameters**

- \text{FRS} \quad \text{Specifies source floating-point register of stored data.}
- \text{D} \quad \text{Specifies a 16-bit signed two's complement integer sign-extended to 32 bits for EA calculation.}
- \text{RA} \quad \text{Specifies source general-purpose register for EA calculation and possible address update.}

**Examples**
The following code stores the doubleword contents of FPR 6 into a location in memory and stores the address in GPR 4:
Related Information

Floating-Point Processor

Floating-Point Load and Store Instructions

stfdux (Store Floating-Point Double with Update Indexed) Instruction

Purpose
Stores a doubleword of data in a specified location in memory and in some cases places the address in a general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>FRS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>759</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

stfdux \[FRS RA RB\]

Description

The stfdux instruction stores the contents of floating-point register (FPR) FRS into the doubleword storage addressed by the effective address (EA).

If general-purpose register (GPR) RA is not 0, the EA is the sum of the contents of GPRs RA and RB. If GPR RA is 0, then the EA is the contents of GPR RB.

If GPR RA does not equal 0 and the storage access does not cause Alignment Interrupt or a Data Storage Interrupt, then the EA is stored in GPR RA.

The stfdux instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

- \(FRS\) Specifies source floating-point register of stored data.
- \(RA\) Specifies source general-purpose register for EA calculation and possible address update.
- \(RB\) Specifies source general-purpose register for EA calculation.
Examples
The following code stores the contents of FPR 6 into a location in memory and stores the address in GPR 4:

```assembly
.csect data[rw]
buffer: .long 0,0,0,0
# Assume FPR 6 contains 0x9000 3000 9000 3000.
# Assume GPR 4 contains 0x0000 0008.
# Assume GPR 5 contains the address of buffer.
.csect text[pr]
stfdx 6,4,5
# buffer+8 now contains 0x9000 3000 9000 3000.
# GPR 4 now contains the address of buffer+8.
```

Related Information
- Floating-Point Processor
- Floating-Point Load and Store Instructions

**stfdx (Store Floating-Point Double Indexed) Instruction**

**Purpose**
Stores a doubleword of data in a specified location in memory.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>FRS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>727</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

```assembly
stfdx FRS RA RB
```

**Description**
The `stfdx` instruction stores the contents of floating-point register (FPR) `FRS` into the doubleword storage addressed by the effective address (EA).

If general-purpose register (GPR) `RA` is not 0, the EA is the sum of the contents of GPRs `RA` and `RB`. If GPR `RA` is 0, then the EA is the contents of GPR `RB`.

The `stfdx` instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

**Parameters**
- **FRS** Specifies source floating-point register of stored data.
- **RA** Specifies source general-purpose register for EA calculation.
- **RB** Specifies source general-purpose register for EA calculation.
Examples
The following code stores the contents of FPR 6 into a location in memory addressed by GPR 5 and GPR 4:

```
csect data[rw]
buffer: .long 0,0,0,0
# Assume FPR 6 contains 0x4865 6C6C 6F20 776F.
# Assume GPR 4 contains 0x0000 0008.
# Assume GPR 5 contains the address of buffer.
csect text[pr]
stdfx 6,4,5
# 0x4865 6C6C 6F20 776F is now stored at the
# address buffer+8.
```

Related Information
Floating-Point Processor.
Floating-Point Load and Store Instructions.

stfiwx (Store Floating-Point as Integer Word Indexed)

Purpose
Stores the low-order 32 bits from a specified floating point register in a specified word location in memory.

Note: The stfiwx instruction is defined only in the PowerPC architecture and is an optional instruction. It is supported on the PowerPC 603 RISC Microprocessor and the PowerPC 604 RISC Microprocessor, but not on the PowerPC 601 RISC Microprocessor.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>FRS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>983</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

```
stfiwx [FRS RA RB]
```

Description
The stfiwx instruction stores the contents of the low-order 32 bits of floating-point register (FPR) FRS, without conversion, into the word storage addressed by the effective address (EA).

If general-purpose register (GPR) RA is not 0, the EA is the sum of the contents of GPRs RA and RB. If GPR RA is 0, then the EA is the contents of GPR RB.

The stfiwx instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

If the contents of register FRS was produced, either directly or indirectly by a Load Floating Point Single Instruction, a single-precision arithmetic instruction, or the frsp (Floating Round to Single Precision)
instruction, then the value stored is undefined. (The contents of FRS is produced directly by such an instruction if FRS is the target register of such an instruction. The contents of register FRS is produced indirectly by such an instruction if FRS is the final target register of a sequence of one or more Floating Point Move Instructions, and the input of the sequence was produced directly by such an instruction.)

**Parameters**

- **FRS** Specifies source floating-point register of stored data.
- **RA** Specifies source general-purpose register for EA calculation.
- **RB** Specifies source general-purpose register for EA calculation.

**Examples**

The following code stores the contents of FPR 6 into a location in memory addressed by GPR 5 and GPR 4:

```assembly
.csect data[rw]
buffer: .long 0,0,0,0
# Assume FPR 6 contains 0x4865 6C6C 6F20 776F.
# Assume GPR 4 contains 0x0000 0008.
# Assume GPR 5 contains the address of buffer.
.csect text[pr]
stfiwx 6,4,5
# 6F20 776F is now stored at the # address buffer+8.
```

**Related Information**

- [Floating-Point Processor](#)
- [Floating-Point Load and Store Instructions](#)

---

**stfq (Store Floating-Point Quad) Instruction**

**Purpose**

Stores in memory two double-precision values at two consecutive doubleword locations.

**Note:** The stfq instruction is supported only in the POWER2 implementation of the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>60</td>
</tr>
<tr>
<td>6-10</td>
<td>FRS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-29</td>
<td>DS</td>
</tr>
<tr>
<td>30-31</td>
<td>00</td>
</tr>
</tbody>
</table>

**POWER2**

```
stfq FRS [DS] [RA]
```
Description
The stfq instruction stores in memory the contents of two consecutive floating-point registers (FPR) at the location specified by the effective address (EA).

DS is sign-extended to 30 bits and concatenated on the right with b'00' to form the offset value. If general-purpose register (GPR) RA is 0, the offset value is the EA. If GPR RA is not 0, the offset value is added to GPR RA to generate the EA. The contents of FPR FRS is stored into the doubleword of storage at the EA. If FPR FRS is 31, then the contents of FPR 0 is stored into the doubleword at EA+8; otherwise, the contents of FRS+1 are stored into the doubleword at EA+8.

The stfq instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters
FRS Specifies the first of two floating-point registers that contain the values to be stored.
DS Specifies a 14-bit field used as an immediate value for the EA calculation.
RA Specifies one source general-purpose register for the EA calculation.

Related Information
The lfqux (Load Floating-Point Quad with Update Indexed) instruction.
Floating-Point Processor.
Floating-Point Load and Store Instructions.

stfq (Store Floating-Point Quad with Update) Instruction

Purpose
Stores in memory two double-precision values at two consecutive doubleword locations and updates the address base.

Note: The stfq instruction is supported only in the POWER2 implementation of the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>61</td>
</tr>
<tr>
<td>6-10</td>
<td>FRS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-29</td>
<td>DS</td>
</tr>
<tr>
<td>30-31</td>
<td>01</td>
</tr>
</tbody>
</table>

POWER2
stfq

FRS [DS | RA]
Description
The stfqu instruction stores in memory the contents of two consecutive floating-point registers (FPR) at the location specified by the effective address (EA).

DS is sign-extended to 30 bits and concatenated on the right with b'00' to form the offset value. If general-purpose register (GPR) RA is 0, the offset value is the EA. If GPR RA is not 0, the offset value is added to GPR RA to generate the EA. The contents of FPR FRS are stored into the doubleword of storage at the EA. If FPR FRS is 31, then the contents of FPR 0 is stored into the doubleword at EA+8; otherwise, the contents of FRS+1 is stored into the doubleword at EA+8.

If GPR RA is not 0, the EA is placed into GPR RA.

The stfqu instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters
FRS Specifies the first of two floating-point registers that contain the values to be stored.
DS Specifies a 14-bit field used as an immediate value for the EA calculation.
RA Specifies one source general-purpose register for the EA calculation and the target register for the EA update.

Related Information
The lfuqx (Load Floating-Point Quad with Update Indexed) instruction.
Floating-Point Processor
Floating-Point Load and Store Instructions

stfqu (Store Floating-Point Quad with Update Indexed) Instruction

Purpose
Stores in memory two double-precision values at two consecutive doubleword locations and updates the address base.

Note: The stfuq instruction is supported only in the POWER2 implementation of the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>FRS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>951</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

POWER2 stfqu FRS RA RB
**Description**

The `stfqx` instruction stores in memory the contents of two consecutive floating-point registers (FPR) at the location specified by the effective address (EA).

If general-purpose register (GPR) `RA` is not 0, the EA is the sum of the contents of GPR `RA` and GPR `RB`. If GPR `RA` is 0, the EA is the contents of GPR `RB`. The contents of FPR `FRS` is stored into the doubleword of storage at the EA. If FPR `FRS` is 31, then the contents of FPR 0 is stored into the doubleword at EA+8; otherwise, the contents of `FRS+1` is stored into the doubleword at EA+8.

If GPR `RA` is not 0, the EA is placed into GPR `RA`.

The `stfqx` instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

**Parameters**

- **FRS**: Specifies the first of two floating-point registers that contain the values to be stored.
- **RA**: Specifies the first source general-purpose register for the EA calculation and the target register for the EA update.
- **RB**: Specifies the second source general-purpose register for the EA calculation.

**Related Information**

The `lfqux` (Load Floating-Point Quad with Update Indexed) instruction.

[Floating-Point Processor](#).

[Floating-Point Load and Store Instructions](#).

**stfqx (Store Floating-Point Quad Indexed) Instruction**

**Purpose**

Stores in memory two double-precision values at two consecutive doubleword locations.

**Note:** The `stfqx` instruction is supported only in the POWER2 implementation of the POWER family architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>FRS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>919</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**POWER2**

`stfqx` `FRS RA RB`
Description
The stfqx instruction stores in memory the contents of floating-point register (FPR) FRS at the location specified by the effective address (EA).

If general-purpose register (GPR) RA is not 0, the EA is the sum of the contents of GPR RA and GPR RB. If GPR RA is 0, the EA is the contents of GPR RB. The contents of FPR FRS is stored into the doubleword of storage at the EA. If FPR FRS is 31, then the contents of FPR 0 is stored into the doubleword at EA+8; otherwise, the contents of FRS+1 is stored into the doubleword at EA+8.

The stfqx instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters
FRS Specifies the first of two floating-point registers that contain the values to be stored.
RA Specifies one source general-purpose register for the EA calculation.
RB Specifies the second source general-purpose register for the EA calculation.

Related Information
The lfqux (Load Floating-Point Quad with Update Indexed) instruction.
Floating-Point Processor.
Floating-Point Load and Store Instructions.

stfs (Store Floating-Point Single) Instruction
Purpose
Stores a word of data from a floating-point register into a specified location in memory.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>52</td>
</tr>
<tr>
<td>6-10</td>
<td>FRS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

stfs FRS D RA

Description
The stfs instruction converts the contents of floating-point register (FPR) FRS to single-precision and stores the result into the word of storage addressed by the effective address (EA).

If general-purpose register (GPR) RA is not 0, the EA is the sum of the contents of GPR RA and D, a 16-bit signed two’s complement integer sign-extended to 32 bits. If GPR RA is 0, then the EA is D.

The stfs instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.
Parameters

**FRS**  
Specifies floating-point register of stored data.

**D**  
Specifies a 16-bit, signed two’s complement integer sign-extended to 32 bits for EA calculation.

**RA**  
Specifies source general-purpose register for EA calculation.

Examples

The following code stores the single-precision contents of FPR 6 into a location in memory:

```plaintext
.csect data[rw]
buffer: .long 0
# Assume FPR 6 contains 0x4865 6C6C 6F20 776F.
# Assume GPR 4 contains the address of csect data[rw].
.csect text[pr]
stfs 6,buffer(4)
# buffer now contains 0x432B 6363.
```

Related Information

- [Floating-Point Processor](#)
- [Floating-Point Load and Store Instructions](#)

**stfsu (Store Floating-Point Single with Update) Instruction**

**Purpose**

Stores a word of data from a floating-point register into a specified location in memory and possibly places the address in a general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>53</td>
</tr>
<tr>
<td>6-10</td>
<td>FRS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

```plaintext
stfsu  FRS D RA
```

**Description**

The **stfsu** instruction converts the contents of floating-point register (FPR) **FRS** to single-precision and stores the result into the word of storage addressed by the effective address (EA).

If general-purpose register (GPR) **RA** is not 0, the EA is the sum of the contents of GPR **RA** and **D**, a 16-bit signed two’s complement integer sign-extended to 32 bits. If GPR **RA** is 0, then the EA is **D**.

If GPR **RA** does not equal 0 and the storage access does not cause Alignment Interrupt or Data Storage Interrupt, then the EA is stored in GPR **RA**.

The **stfsu** instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.
Parameters

FRS  Specifies floating-point register of stored data.
D    Specifies a 16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
RA   Specifies source general-purpose register for EA calculation and possible address update.

Examples
The following code stores the single-precision contents of FPR 6 into a location in memory and stores the address in GPR 4:

```
csect data[rw]
buffer: .long 0
# Assume FPR 6 contains 0x4865 6C6C 6F20 776F.
# Assume GPR 4 contains the address of csect data[rw].
csect text[pr]
stfs 6,buffer(4)
# GPR 4 now contains the address of buffer.
# buffer now contains 0x432B 6363.
```

Related Information
Floating-Point Processor.
Floating-Point Load and Store Instructions.

stfsux (Store Floating-Point Single with Update Indexed) Instruction

Purpose
Stores a word of data from a floating-point register into a specified location in memory and possibly places the address in a general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>FRS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>695</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

```
stfsux FRS RA RB
```

Description
The `stfsux` instruction converts the contents of floating-point register (FPR) `FRS` to single-precision and stores the result into the word of storage addressed by the effective address (EA).

If general-purpose register (GPR) `RA` is not 0, the EA is the sum of the contents of GPR `RA` and GPR `RB`. If GPR `RA` is 0, then the EA is the contents of GPR `RB`.

If GPR `RA` does not equal 0 and the storage access does not cause Alignment Interrupt or Data Storage Interrupt, then the EA is stored in GPR `RA`.

416  Assembler Language Reference
The **stfsux** instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

**Parameters**

- **FRS** Specifies floating-point register of stored data.
- **RA** Specifies source general-purpose register for EA calculation and possible address update.
- **RB** Specifies source general-purpose register for EA calculation.

**Examples**

The following code stores the single-precision contents of FPR 6 into a location in memory and stores the address in GPR 5:

```assembly
.csect data[rw]
buffer: .long 0,0,0,0
# Assume GPR 4 contains 0x0000 0008.
# Assume GPR 5 contains the address of buffer.
# Assume FPR 6 contains 0x4865 6C6C 6F20 776F.
.csect text[pr]
stfsux 6,5,4
# GPR 5 now contains the address of buffer+8.
# buffer+8 contains 0x432B 6363.
```

**Related Information**

- [Floating-Point Processor](#)
- [Floating-Point Load and Store Instructions](#)

---

**stfsx (Store Floating-Point Single Indexed) Instruction**

**Purpose**

Stores a word of data from a floating-point register into a specified location in memory.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td><strong>FRS</strong></td>
</tr>
<tr>
<td>11-15</td>
<td><strong>RA</strong></td>
</tr>
<tr>
<td>16-20</td>
<td><strong>RB</strong></td>
</tr>
<tr>
<td>21-30</td>
<td>663</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

**Description**

The **stfsx** instruction converts the contents of floating-point register (FPR) **FRS** to single-precision and stores the result into the word of storage addressed by the effective address (EA).

If general-purpose register (GPR) **RA** is not 0, the EA is the sum of the contents of GPR **RA** and GPR **RB**. If GPR **RA** is 0, then the EA is the contents of GPR **RB**.
The **stfsx** instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

**Parameters**

- **FRS** Specifies source floating-point register of stored data.
- **RA** Specifies source general-purpose register for EA calculation.
- **RB** Specifies source general-purpose register for EA calculation.

**Examples**

The following code stores the single-precision contents of FPR 6 into a location in memory:

```
csect data[rw]
buffer: .long 0
# Assume FPR 6 contains 0x4865 6C6C 6F20 776F.
# Assume GPR 4 contains the address of buffer.
csect text[pr]
stfsx 6,0,4
# buffer now contains 0x432B 6363.
```

**Related Information**

- [Floating-Point Processor](#)
- [Floating-Point Load and Store Instructions](#)

### sth (Store Half) Instruction

**Purpose**

Stores a halfword of data from a general-purpose register into a specified location in memory.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>44</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

```
sth RS D RA
```

**Description**

The **sth** instruction stores bits 16-31 of general-purpose register (GPR) **RS** into the halfword of storage addressed by the effective address (EA).

If GPR **RA** is not 0, the EA is the sum of the contents of GPR **RA** and **D**, a 16-bit signed two’s complement integer sign-extended to 32 bits. If GPR **RA** is 0, then the EA is **D**.

The **sth** instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.
Parameters

RS  Specifies source general-purpose register of stored data.
D   Specifies a 16-bit signed two's complement integer sign-extended to 32 bits for EA calculation.
RA  Specifies source general-purpose register for EA calculation.

Examples

The following code stores bits 16-31 of GPR 6 into a location in memory:

```assembly
.csect data[rw]
buffer: .long 0
# Assume GPR 4 contains the address of csect data[rw].
# Assume GPR 6 contains 0x9000 3000.
.csect text[pr]
sth 6,buffer(4)
# buffer now contains 0x3000.
```

Related Information

- Floating-Point Processor.
- Floating-Point Load and Store Instructions.

sthbrx (Store Half Byte-Reverse Indexed) Instruction

Purpose

Stores a halfword of data from a general-purpose register into a specified location in memory with the two bytes reversed.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>918</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

**sthbrx**  **RS**  **RA**  **RB**

Description

The **sthbrx** instruction stores bits 16-31 of general-purpose register (GPR) **RS** into the halfword of storage addressed by the effective address (EA).

Consider the following when using the **sthbrx** instruction:

- Bits 24-31 of GPR **RS** are stored into bits 00-07 of the halfword in storage addressed by EA.
- Bits 16-23 of GPR **RS** are stored into bits 08-15 of the word in storage addressed by EA.

If GPR **RA** is not 0, the EA is the sum of the contents of GPR **RA** and GPR **RB**. If GPR **RA** is 0, then the EA is the contents of GPR **RB**.
The `sthbx` instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Parameters**

- **RS** Specifies source general-purpose register of stored data.
- **RA** Specifies source general-purpose register for EA calculation.
- **RB** Specifies source general-purpose register for EA calculation.

**Examples**

The following code stores the halfword contents of GPR 6 with the bytes reversed into a location in memory:

```assembly
.csect data[rw]
buffer: .long 0
# Assume GPR 6 contains 0x9000 3456.
# Assume GPR 4 contains the address of buffer.
.csect text[pr]
sthbx 6,0,4
# buffer now contains 0x5634.
```

**Related Information**

- Floating-Point Processor
- Floating-Point Load and Store Instructions

**sthu (Store Half with Update) Instruction**

**Purpose**

Stores a halfword of data from a general-purpose register into a specified location in memory and possibly places the address in another general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>45</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

**Description**

The `sthu` instruction stores bits 16-31 of general-purpose register (GPR) `RS` into the halfword of storage addressed by the effective address (EA).

If GPR `RA` is not 0, the EA is the sum of the contents of GPR `RA` and `D`, a 16-bit signed two’s complement integer sign-extended to 32 bits. If GPR `RA` is 0, then the EA is `D`.

If GPR `RA` does not equal 0 and the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, then the EA is placed into GPR `RA`. 420
The **sthu** instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Parameters**

*RS*  
Specifies source general-purpose register of stored data.

*D*  
Specifies a 16-bit signed two's complement integer sign-extended to 32 bits for EA calculation.

*RA*  
Specifies source general-purpose register for EA calculation and possible address update.

**Examples**

The following code stores the halfword contents of GPR 6 into a memory location and stores the address in GPR 4:

```assembly
.data
  buffer: .long 0
  # Assume GPR 6 contains 0x9000 3456.
  # Assume GPR 4 contains the address of csect data[rw].
.text
  sthu 6, buffer(4)

  # buffer now contains 0x3456
  # GPR 4 contains the address of buffer.
```

**Related Information**

[Fixed-Point Processor](#).

[Fixed-Point Load and Store with Update Instructions](#).

---

**sthux (Store Half with Update Indexed) Instruction**

**Purpose**

Stores a halfword of data from a general-purpose register into a specified location in memory and possibly places the address in another general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>439</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

**Description**

The **sthux** instruction stores bits 16-31 of general-purpose register (GPR) *RS* into the halfword of storage addressed by the effective address (EA).

If GPR *RA* is not 0, the EA is the sum of the contents of GPR *RA* and GPR *RB*. If GPR *RA* is 0, then the EA is the contents of GPR *RB*. 

---

Chapter 8. Instruction Set 421
If GPR $RA$ does not equal 0 and the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, then the EA is placed into register GPR $RA$.

The **sthux** instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Parameters**

$RS$  
Specifies source general-purpose register of stored data.

$RA$  
Specifies source general-purpose register for EA calculation and possible address update.

$RB$  
Specifies source general-purpose register for EA calculation.

**Examples**

The following code stores the halfword contents of GPR 6 into a memory location and stores the address in GPR 4:

```assembly
csect data[rw]
buffer: .long 0,0,0,0
# Assume GPR 6 contains 0x9000 3456.
# Assume GPR 4 contains 0x0000 0007.
# Assume GPR 5 contains the address of buffer.
csect text[pr]
sthux 6,4,5
# buffer+0x07 contains 0x3456.
# GPR 4 contains the address of buffer+0x07.
```

**Related Information**

- Fixed-Point Processor
- Fixed-Point Load and Store with Update Instructions

---

**sthx (Store Half Indexed) Instruction**

**Purpose**

Stores a halfword of data from a general-purpose register into a specified location in memory.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>$RS$</td>
</tr>
<tr>
<td>11-15</td>
<td>$RA$</td>
</tr>
<tr>
<td>16-20</td>
<td>$RB$</td>
</tr>
<tr>
<td>21-30</td>
<td>407</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

**sthx**  
$RS$ $RA$ $RB$

**Description**

The **sthx** instruction stores bits 16-31 of general-purpose register (GPR) $RS$ into the halfword of storage addressed by the effective address (EA).
If GPR RA is not 0, the EA is the sum of the contents of GPR RA and GPR RB. If GPR RA is 0, then the EA is the contents of GPR RB.

The sthx instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

| RS | Specifies source general-purpose register of stored data. |
| RA | Specifies source general-purpose register for EA calculation. |
| RB | Specifies source general-purpose register for EA calculation. |

Examples

The following code stores halfword contents of GPR 6 into a location in memory:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 6 contains 0x9000 3456.
# Assume GPR 5 contains the address of buffer.
.csect text[pr]
sthx 6,0,5
# buffer now contains 0x3456.
```

Related Information

- Fixed-Point Processor
- Fixed-Point Load and Store Instructions

### stmw or stm (Store Multiple Word) Instruction

#### Purpose

Stores the contents of consecutive registers into a specified memory location.

#### Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>47</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

**PowerPC**

stmw  

**POWER family**

stm

#### Description

The stmw and stm instructions store $N$ consecutive words from general-purpose register (GPR) RS through GPR 31. Storage starts at the effective address (EA). $N$ is a register number equal to 32 minus RS.
If GPR RA is not 0, the EA is the sum of the contents of GPR RA and D. The sum is a 16-bit signed two’s complement integer sign-extended to 32 bits. If GPR RA is 0, then the EA is D.

The **stmw** instruction has one syntax form. If the EA is not a multiple of 4, the results are boundedly undefined.

The **stm** instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Parameters**

- **RS** Specifies source general-purpose register of stored data.
- **D** Specifies a 16-bit signed two’s complement integer sign-extended to 32 bits for EA calculation.
- **RA** Specifies source general-purpose register for EA calculation.

**Examples**

The following code stores the contents of GPR 29 through GPR 31 into a location in memory:

```
csect data[rw]
buffer: .long 0,0,0
# Assume GPR 29 contains 0x1000 2200.
# Assume GPR 30 contains 0x1000 3300.
# Assume GPR 31 contains 0x1000 4400.
csect text[pr]
stmw 29,buffer(4)
# Three consecutive words in storage beginning at the address
# of buffer are now 0x1000 2200 1000 3300 1000 4400.
```

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Load and Store Instructions](#)

---

**stq** (Store Quad Word) Instruction

**Purpose**

Store a quad-word of data from a general purpose register into a specified memory location.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>62</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-29</td>
<td>DS</td>
</tr>
<tr>
<td>30-31</td>
<td>2</td>
</tr>
</tbody>
</table>

**PowerPC 64**

- **stq** "RS" on page 425 "DS" on page 425 "RA" on page 425
Description

The **stq** instruction stores a quad-word in storage from the source general-purpose registers (GPR) **RS** and **RS+1** into the specified location in memory referenced by the effective address (EA).

If GPR **RA** is not 0, the EA is the sum of the contents of GPR **RA** and **DS**, a 14-bit, signed two's complement integer, which is concatenated on the right by 0b00 and sign extended to 64-bits. If GPR **RA** is 0, then the EA is **DS**.

**Parameters**

- **RS** Specifies the source general-purpose register containing data. If **RS** is odd, the instruction form is invalid.
- **DS** Specifies a 14-bit, signed two's complement integer which is concatenated on the right with 0b00 and sign-extended to 64 bits for EA calculation.
- **RA** Specifies source general-purpose register for EA calculation.

**Implementation**

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

**Related Information**

- “Fixed-Point Processor” on page 21.
- “Fixed-Point Load and Store Instructions” on page 21.

**stswi or stsi (Store String Word Immediate) Instruction**

**Purpose**

Stores consecutive bytes from consecutive registers into a specified location in memory.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td><strong>RS</strong></td>
</tr>
<tr>
<td>11-15</td>
<td><strong>RA</strong></td>
</tr>
<tr>
<td>16-20</td>
<td><strong>NB</strong></td>
</tr>
<tr>
<td>21-30</td>
<td>725</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

**PowerPC**

**stswi**

**POWER family**

**stsi**

**Description**

The **stswi** and **stsi** instructions store **N** consecutive bytes starting with the leftmost byte in general-purpose register (GPR) **RS** at the effective address (EA) from GPR **RS** through GPR **RS + NR - 1**.
If GPR RA is not 0, the EA is the contents of GPR RA. If RA is 0, then the EA is 0.

Consider the following when using the stswi and stsi instructions:
- $NB$ is the byte count.
- $RS$ is the starting register.
- $N$ is $NB$, which is the number of bytes to store. If $NB$ is 0, then $N$ is 32.
- $NR$ is ceiling($N$/4), which is the number of registers to store data from.

For the POWER family instruction stsi, the contents of the MQ Register are undefined.

The stswi and stsi instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Parameters**

- **RS** Specifies source general-purpose register of stored data.
- **RA** Specifies source general-purpose register for EA calculation.
- **NB** Specifies byte count for EA calculation.

**Examples**

The following code stores the bytes contained in GPR 6 to GPR 8 into a location in memory:

```assembly
.csect data[ rw]
buffer: .long 0,0,0
# Assume GPR 4 contains the address of buffer.
# Assume GPR 6 contains 0x4865 6C6C.
# Assume GPR 7 contains 0x6F20 776F.
# Assume GPR 8 contains 0x726C 6421.
.csect text[ pr]
stswi 6,4,12  # buffer now contains 0x4865 6C6C 6F20 776F 726C 6421.
```

**Related Information**

- Fixed-Point Processor
- Fixed-Point String Instructions

### stswx or stsx (Store String Word Indexed) Instruction

**Purpose**
Stores consecutive bytes from consecutive registers into a specified location in memory.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>661</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>
Description

The **stswx** and **stsx** instructions store *N* consecutive bytes starting with the leftmost byte in register *RS* at the effective address (EA) from general-purpose register (GPR) *RS* through GPR *RS* + *NR* - 1.

If GPR *RA* is not 0, the EA is the sum of the contents of GPR *RA* and the contents of GPR *RB*. If GPR *RA* is 0, then EA is the contents of GPR *RB*.

Consider the following when using the **stswx** and **stsx** instructions:
- *XER25-31* contain the byte count.
- *RS* is the starting register.
- *N* is *XER25-31*, which is the number of bytes to store.
- *NR* is ceiling(*N*/4), which is the number of registers to store data from.

For the POWER family instruction **stsx**, the contents of the MQ Register are undefined.

The **stswx** and **stsx** instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

- **RS** Specifies source general-purpose register of stored data.
- **RA** Specifies source general-purpose register for EA calculation.
- **RB** Specifies source general-purpose register for EA calculation.

Examples

The following code stores the bytes contained in GPR 6 to GPR 7 into the specified bytes of a location in memory:

```plaintext
.csect data[rw]
buffer: .long 0,0,0
# Assume GPR 5 contains 0x0000 0007.
# Assume GPR 4 contains the address of buffer.
# Assume GPR 6 contains 0x4865 6C6C.
# Assume GPR 7 contains 0x6F20 776F.
# The Fixed-Point Exception Register bits 25-31 contain 6.
.csect text[pr]
stswx 6,4,5
# buffer+0x7 now contains 0x4865 6C6C 6F20.
```

Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point String Instructions](#)
stw or st (Store) Instruction

Purpose
Stores a word of data from a general-purpose register into a specified location in memory.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>36</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

PowerPC
stw  

POWER family
st  

Description
The stw and st instructions store a word from general-purpose register (GPR) RS into a word of storage addressed by the effective address (EA).

If GPR RA is not 0, the EA is the sum of the contents of GPR RA and D, a 16-bit signed two’s complement integer sign-extended to 32 bits. If GPR RA is 0, then the EA is D.

The stw and st instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>Specifies source general-purpose register of stored data.</td>
</tr>
<tr>
<td>D</td>
<td>Specifies a 16-bit signed two’s complement integer sign-extended to 32 bits for EA calculation.</td>
</tr>
<tr>
<td>RA</td>
<td>Specifies source general-purpose register for EA calculation.</td>
</tr>
</tbody>
</table>

Examples
The following code stores the contents of GPR 6 into a location in memory:

```
.csect data[rw]
buffer:.long 0,0
# Assume GPR 6 contains 0x9000 3000.
# Assume GPR 5 contains the address of buffer.
.csect text[pr]
stw 6,4(5)
# 0x9000 3000 is now stored at the address buffer+4.
```

Related Information

Fixed-Point Processor.

Fixed-Point Load and Store Instructions.
stwbrx or stbrx (Store Word Byte-Reverse Indexed) Instruction

Purpose
Stores a byte-reversed word of data from a general-purpose register into a specified location in memory.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>662</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

PowerPC
stwbrx \( RS, RA, RB \)

POWER family
stbrx \( RS, RA, RB \)

Description
The stwbrx and stbrx instructions store a byte-reversed word from general-purpose register (GPR) \( RS \) into a word of storage addressed by the effective address (EA).

Consider the following when using the stwbrx and stbrx instructions:
- Bits 24-31 of GPR \( RS \) are stored into bits 00-07 of the word in storage addressed by EA.
- Bits 16-23 of GPR \( RS \) are stored into bits 08-15 of the word in storage addressed by EA.
- Bits 08-15 of GPR \( RS \) are stored into bits 16-23 of the word in storage addressed by EA.
- Bits 00-07 of GPR \( RS \) are stored into bits 24-31 of the word in storage addressed by EA.

If GPR \( RA \) is not 0, the EA is the sum of the contents of GPR \( RA \) and GPR \( RB \). If GPR \( RA \) is 0, then the EA is the contents of GPR \( RB \).

The stwbrx and stbrx instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters
- \( RS \) Specifies source general-purpose register of stored data.
- \( RA \) Specifies source general-purpose register for EA calculation.
- \( RB \) Specifies source general-purpose register for EA calculation.

Examples
The following code stores a byte-reverse word from GPR 6 into a location in memory:
```assembly
.csect data[rw]
buffer: .long 0
# Assume GPR 4 contains the address of buffer.
```
Related Information

Fixed-Point Processor.

Fixed-Point Load and Store Instructions.

stwcx. (Store Word Conditional Indexed) Instruction

Purpose

Used in conjunction with a preceding lwarx instruction to emulate a read-modify-write operation on a specified memory location.

Note: The stwcx. instruction is supported only in the PowerPC architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>150</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
</tr>
</tbody>
</table>

PowerPC stwcx. RS RA RB

Description

The stwcx. and lwarx instructions are primitive, or simple, instructions used to perform a read-modify-write operation to storage. If the store is performed, the use of the stwcx. and lwarx instructions ensures that no other processor or mechanism has modified the target memory location between the time the lwarx instruction is executed and the time the stwcx. instruction completes.

Consider the following when using the stwcx. instruction:

- If general-purpose register (GPR) RA is 0, the effective address (EA) is the content of GPR RB, otherwise EA is the sum of the content of GPR RA plus the content of GPR RB.
- If the reservation created by a lwarx instruction exists, the content of GPR RS is stored into the word in storage and addressed by EA and the reservation is cleared. Otherwise, the storage is not altered.
- If the store is performed, bits 0-2 of Condition Register Field 0 are set to 0b001, otherwise, they are set to 0b000. The SO bit of the XER is copied to to bit 4 of Condition Register Field 0.

The stwcx instruction has one syntax form and does not affect the Fixed-Point Exception Register. If the EA is not a multiple of 4, the results are boundedly undefined.
Parameters

RS  Specifies source general-purpose register of stored data.
RA  Specifies source general-purpose register for EA calculation.
RB  Specifies source general-purpose register for EA calculation.

Examples

1. The following code performs a “Fetch and Store” by atomically loading and replacing a word in storage:

   # Assume that GPR 4 contains the new value to be stored.
   # Assume that GPR 3 contains the address of the word
   # to be loaded and replaced.
   loop: lwarx r5,0,r3  # Load and reserve
       stwcx. r4,0,r3  # Store new value if still
                   # reserved
       bne-  loop     # Loop if lost reservation
   # The new value is now in storage.
   # The old value is returned to GPR 4.

2. The following code performs a “Compare and Swap” by atomically comparing a value in a register with a word in storage:

   # Assume that GPR 5 contains the new value to be stored after
   # a successful match.
   # Assume that GPR 3 contains the address of the word
   # to be tested.
   # Assume that GPR 4 contains the value to be compared against
   # the value in memory.
   loop: lwarxr 6,0,r3  # Load and reserve
       cmpw  r4,r6     # Are the first two operands
                   # equal?
       bne-  exit      # Skip if not equal
       stwcx. r5,0,r3  # Store new value if still
                       # reserved
       bne-  loop      # Loop if lost reservation
exit: mrr  4,r6    # Return value from storage
   # The old value is returned to GPR 4.
   # If a match was made, storage contains the new value.

If the value in the register equals the word in storage, the value from a second register is stored in the word in storage. If they are unequal, the word from storage is loaded into the first register and the EQ bit of the Condition Register Field 0 is set to indicate the result of the comparison.

Related Information

The lwarx (Load Word and Reserve Indexed) instruction.

Processing and Storage

stwu or stu (Store Word with Update) Instruction

Purpose

Stores a word of data from a general-purpose register into a specified location in memory and possibly places the address in another general-purpose register.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>37</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>D</td>
</tr>
</tbody>
</table>

PowerPC

```
stwu   RS D RA
```

POWER family

```
stu    RS D RA
```

Description
The **stwu** and **stu** instructions store the contents of general-purpose register (GPR) **RS** into the word of storage addressed by the effective address (EA).

If GPR **RA** is not 0, the EA is the sum of the contents of GPR **RA** and **D**, a 16-bit signed two’s complement integer sign-extended to 32 bits. If GPR **RA** is 0, then the EA is **D**.

If GPR **RA** is not 0 and the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, then EA is placed into GPR **RA**.

The **stwu** and **stu** instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

- **RS** Specifies general-purpose register of stored data.
- **D** Specifies 16-bit signed two’s complement integer sign-extended to 32 bits for EA calculation.
- **RA** Specifies source general-purpose register for EA calculation and possible address update.

Examples
The following code stores the contents of GPR 6 into a location in memory:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 4 contains the address of csect data[rw].
# Assume GPR 6 contains 0x9000 3000.
.csect text[pr]
stwu 6,buffer(4)
# buffer now contains 0x9000 3000.
# GPR 4 contains the address of buffer.
```

Related Information
- [Fixed-Point Processor](#)
- [Fixed-Point Load and Store with Update Instructions](#)
stwux or stux (Store Word with Update Indexed) Instruction

**Purpose**
Stores a word of data from a general-purpose register into a specified location in memory and possibly places the address in another general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>212-30</td>
<td>183</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

- **PowerPC**
  - `stwux`  
  - `stux`  

- **POWER family**
  - `stwux`  
  - `stux`

**Description**
The `stwux` and `stux` instructions store the contents of general-purpose register (GPR) `RS` into the word of storage addressed by the effective address (EA).

If GPR `RA` is not 0, the EA is the sum of the contents of GPR `RA` and GPR `RB`. If GPR `RA` is 0, then the EA is the contents of GPR `RB`.

If GPR `RA` is not 0 and the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, then the EA is placed into GPR `RA`.

The `stwux` and `stux` instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Parameters**
- **RS**  
  - Specifies source general-purpose register of stored data.
- **RA**  
  - Specifies source general-purpose register for EA calculation and possible address update.
- **RB**  
  - Specifies source general-purpose register for EA calculation.

**Examples**
The following code stores the contents of GPR 6 into a location in memory:

```assembly
.csect data[rw]
buffer: .long 0,0
# Assume GPR 4 contains 0x0000 0004.
# Assume GPR 23 contains the address of buffer.
# Assume GPR 6 contains 0x9000 3000.
```
stwx 6,4,23
# buffer+4 now contains 0x9000 3000.
# GPR 4 now contains the address of buffer+4.

Related Information

- Fixed-Point Processor
- Fixed-Point Load and Store with Update Instructions

stwx or stx (Store Word Indexed) Instruction

Purpose
Stores a word of data from a general-purpose register into a specified location in memory.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>151</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

PowerPC
stwx

POWER family
stx

Description
The stwx and stx instructions store the contents of general-purpose register (GPR) RS into the word of storage addressed by the effective address (EA).

If GPR RA is not 0, the EA is the sum of the contents of GPR RA and GPR RB. If GPR RA is 0, then the EA is the contents of GPR RB.

The stwx and stx instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

- **RS** Specifies source general-purpose register of stored data.
- **RA** Specifies source general-purpose register for EA calculation.
- **RB** Specifies source general-purpose register for EA calculation.

Examples
The following code stores the contents of GPR 6 into a location in memory:
.csect data[pr]
buffer: .long 0
# Assume GPR 4 contains the address of buffer.
# Assume GPR 6 contains 0x4865 6C6C.
.csect text[pr]
stwx 6,0,4
# Buffer now contains 0x4865 6C6C.

Related Information

Fixed-Point Processor.

Fixed-Point Load and Store Instructions.

**subf (Subtract From) Instruction**

**Purpose**
Subtracts the contents of two general-purpose registers and places the result in a third general-purpose register.

*Note*: The subf instruction is supported only in the PowerPC architecture.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>40</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**PowerPC**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>subf</td>
<td>RT, RA, RB</td>
</tr>
<tr>
<td>subf.</td>
<td>RT, RA, RB</td>
</tr>
<tr>
<td>subfo</td>
<td>RT, RA, RB</td>
</tr>
<tr>
<td>subfo.</td>
<td>RT, RA, RB</td>
</tr>
</tbody>
</table>

See [Extended Mnemonics of Fixed-Point Arithmetic Instructions](#) for more information.

**Description**
The subf instruction adds the ones complement of the contents of general-purpose register (GPR) RA and 1 to the contents of GPR RB and stores the result in the target GPR RT.

The subf instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>subf</td>
<td>0</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>
The four syntax forms of the **subf** instruction never affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

### Parameters

**RT**  
Specifies target general-purpose register where result of operation is stored.

**RA**  
Specifies source general-purpose register for EA calculation.

**RB**  
Specifies source general-purpose register for EA calculation.

### Examples

1. The following code subtracts the contents of GPR 4 from the contents of GPR 10, and stores the result in GPR 6:
   
   ```
   # Assume GPR 4 contains 0x8000 7000.
   # Assume GPR 10 contains 0x9000 3000.
   subf 6,4,10
   # GPR 6 now contains 0x0FFF C000.
   ```

2. The following code subtracts the contents of GPR 4 from the contents of GPR 10, stores the result in GPR 6, and sets Condition Register Field 0:
   
   ```
   # Assume GPR 4 contains 0x0000 4500.
   # Assume GPR 10 contains 0x8000 7000.
   subf 6,4,10
   # GPR 6 now contains 0x8000 2800.
   ```

3. The following code subtracts the contents of GPR 4 from the contents of GPR 10, stores the result in GPR 6, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register to reflect the result of the operation:
   
   ```
   # Assume GPR 4 contains 0x8000 0000.
   # Assume GPR 10 contains 0x0000 4500.
   subfo 6,4,10
   # GPR 6 now contains 0x8000 4500.
   ```

4. The following code subtracts the contents of GPR 4 from the contents of GPR 10, stores the result in GPR 6, and sets the Summary Overflow and Overflow bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:
   
   ```
   # Assume GPR 4 contains 0x8000 0000.
   # Assume GPR 10 contains 0x0000 7000.
   subfo 6,4,10
   # GPR 6 now contains 0x8000 7000.
   ```

### Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Arithmetic Instructions](#)
**subfc or sf (Subtract from Carrying) Instruction**

**Purpose**
Subtracts the contents of a general-purpose register from the contents of another general-purpose register and places the result in a third general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>8</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**PowerPC**

- **subfc**
  - `RT RA RB`

- **subfc.**
  - `RT RA RB`

- **subfco**
  - `RT RA RB`

- **subfco.**
  - `RT RA RB`

**POWER family**

- **sf**
  - `RT RA RB`

- **sf.**
  - `RT RA RB`

- **sfo**
  - `RT RA RB`

- **sfo.**
  - `RT RA RB`

See [Extended Mnemonics of Fixed-Point Arithmetic Instructions](#) for more information.

**Description**
The **subfc** and **sf** instructions add the ones complement of the contents of general-purpose register (GPR) `RA` and 1 to the contents of GPR `RB` and stores the result in the target GPR `RT`.

The **subfc** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

The **sf** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>subfc</td>
<td>0</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>subfc.</td>
<td>0</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>subfco</td>
<td>1</td>
<td>SO,OV,CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>subfco.</td>
<td>1</td>
<td>SO,OV,CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>sf</td>
<td>0</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>
The four syntax forms of the **subfc** instruction, and the four syntax forms of the **sf** instruction, always affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- **RT**  
  Specifies target general-purpose register where result of operation is stored.
- **RA**  
  Specifies source general-purpose register for operation.
- **RB**  
  Specifies source general-purpose register for operation.

**Examples**

1. The following code subtracts the contents of GPR 4 from the contents of GPR 10, stores the result in GPR 6, and sets the Carry bit to reflect the result of the operation:

    ```
    # Assume GPR 4 contains 0x8000 7000.
    # Assume GPR 10 contains 0x9000 3000.
    subfc 6,4,10
    # GPR 6 now contains 0x0FFF C000.
    ```

2. The following code subtracts the contents of GPR 4 from the contents of GPR 10, stores the result in GPR 6, and sets Condition Register Field 0 and the Carry bit to reflect the result of the operation:

    ```
    # Assume GPR 4 contains 0x0000 4500.
    # Assume GPR 10 contains 0x8000 7000.
    subfco 6,4,10
    # GPR 6 now contains 0x8000 2800.
    ```

3. The following code subtracts the contents of GPR 4 from the contents of GPR 10, stores the result in GPR 6, and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register to reflect the result of the operation:

    ```
    # Assume GPR 4 contains 0x8000 0000.
    # Assume GPR 10 contains 0x0000 4500.
    subfco 6,4,10
    # GPR 6 now contains 0x8000 4500.
    ```

4. The following code subtracts the contents of GPR 4 from the contents of GPR 10, stores the result in GPR 6, and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

    ```
    # Assume GPR 4 contains 0x8000 0000.
    # Assume GPR 10 contains 0x0000 7000.
    subfco 6,4,10
    # GPR 6 now contains 0x8000 7000.
    ```

**Related Information**

- [Fixed-Point Processor](#)
- [Fixed-Point Arithmetic Instructions](#)
subfe or sfe (Subtract from Extended) Instruction

Purpose
Adds the one’s complement of the contents of a general-purpose register to the sum of another general-purpose register and then adds the value of the Fixed-Point Exception Register Carry bit and stores the result in a third general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>136</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC
subfe  
subfe. 
subfeo  
subfeo.

POWER family
sfe  
sfe.  
sfeo  
sfeo.

Description
The subfe and sfe instructions add the value of the Fixed-Point Exception Register Carry bit, the contents of general-purpose register (GPR) \( RB \), and the one’s complement of the contents of GPR \( RA \) and store the result in the target GPR \( RT \).

The subfe instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

The sfe instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>subfe</td>
<td>0</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>subfe.</td>
<td>0</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>subfeo</td>
<td>1</td>
<td>SO,OV,CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>subfeo.</td>
<td>1</td>
<td>SO,OV,CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>sfe</td>
<td>0</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>
The four syntax forms of the `subfe` instruction, and the four syntax forms of the `sfe` instruction, always affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- **RT** Specifies target general-purpose register where result of operation is stored.
- **RA** Specifies source general-purpose register for operation.
- **RB** Specifies source general-purpose register for operation.

**Examples**

1. The following code adds the one’s complement of the contents of GPR 4, the contents of GPR 10, and the value of the Fixed-Point Exception Register Carry bit and stores the result in GPR 6:

   ```
   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 10 contains 0x8000 7000.
   # Assume the Carry bit is one.
   subfe 6,4,10
   # GPR 6 now contains 0xF000 4000.
   ```

2. The following code adds the one’s complement of the contents of GPR 4, the contents of GPR 10, and the value of the Fixed-Point Exception Register Carry bit, stores the result in GPR 6, and sets Condition Register field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x0000 4500.
   # Assume GPR 10 contains 0x8000 7000.
   # Assume the Carry bit is zero.
   subfe 6,4,10
   # GPR 6 now contains 0x8000 2AFF.
   ```

3. The following code adds the one’s complement of the contents of GPR 4, the contents of GPR 10, and the value of the Fixed-Point Exception Register Carry bit, stores the result in GPR 6, and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x8000 0000.
   # Assume GPR 10 contains 0xEFFF FFFF.
   # Assume the Carry bit is one.
   subfeo 6,4,10
   # GPR 6 now contains 0x6FFF FFFF.
   ```

4. The following code adds the one’s complement of the contents of GPR 4, the contents of GPR 10, and the value of the Fixed-Point Exception Register Carry bit, stores the result in GPR 6, and sets the Summary Overflow, Overflow, and Carry bits in the Fixed-Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

   ```
   # Assume GPR 4 contains 0x8000 0000.
   # Assume GPR 10 contains 0xEFFF FFFF.
   # Assume the Carry bit is zero.
   subfeo 6,4,10
   # GPR 6 now contains 0x6FFF FFFE.
   ```
Fixed-Point Processor.

Fixed-Point Arithmetic Instructions.

Subtract from Immediate Carrying (subfic or sfi)

**Purpose**
Subtracts the contents of a general-purpose register from a 16-bit signed integer and places the result in another general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>08</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>SI</td>
</tr>
</tbody>
</table>

PowerPC

```
subfic   RT RA SI
```

POWER family

```
sfi      RT RA SI
```

**Description**
The `subfic` and `sfi` instructions add the one’s complement of the contents of general-purpose register (GPR) RA, 1, and a 16-bit signed integer SI. The result is placed in the target GPR RT.

**Note:** When SI is -1, the `subfic` and `sfi` instructions place the one’s complement of the contents of GPR RA in GPR RT.

The `subfic` and `sfi` instructions have one syntax form and do not affect Condition Register Field 0. These instructions always affect the Carry bit in the Fixed-Point Exception Register.

**Parameters**

- **RT** Specifies target general-purpose register where result of operation is stored.
- **RA** Specifies source general-purpose register for operation.
- **SI** Specifies 16-bit signed integer for operation.

**Examples**
The following code subtracts the contents of GPR 4 from the signed integer 0x0000 7000 and stores the result in GPR 6:

```c
# Assume GPR 4 holds 0x9000 3000.
subic 6,4,0x0000 7000
# GPR 6 now holds 0x7000 4000.
```
Related Information

Fixed-Point Processor

Fixed-Point Arithmetic Instructions

subfme or sfme (Subtract from Minus One Extended) Instruction

Purpose
Adds the one’s complement of a general-purpose register to -1 with carry.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>232</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC

subfme \( RT, RA \)
subfme. \( RT, RA \)
sfme \( RT, RA \)
sfmeo \( RT, RA \)

POWER family

sfme \( RT, RA \)
sfme. \( RT, RA \)
sfmeo \( RT, RA \)
sfmeo. \( RT, RA \)

Description
The subfme and sfme instructions add the one’s complement of the contents of general-purpose register(GPR) \( RA \), the Carry Bit of the Fixed-Point Exception Register, and \( x'\text{FFFFFFFF}' \) and place the result in the target GPR \( RT \).

The subfme instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

The sfme instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>subfme</td>
<td>0</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>subfme.</td>
<td>0</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>
The four syntax forms of the **subfme** instruction, and the four syntax forms of the **sfme** instruction, always affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

### Parameters

**RT**  
Specifies target general-purpose register where result of operation is stored.

**RA**  
Specifies source general-purpose register for operation.

### Examples

1. The following code adds the one’s complement of the contents of GPR 4, the Carry bit of the Fixed-Point Exception Register, and x’FFFFFFF’ and stores the result in GPR 6:
   
   ```
   subfme 6,4
   
   # Assume GPR 4 contains 0x9000 3000.
   # Assume the Carry bit is set to one.
   # GPR 6 now contains 0x6FFF CFFF.
   ```

2. The following code adds the one’s complement of the contents of GPR 4, the Carry bit of the Fixed-Point Exception Register, and x’FFFFFFF’, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:
   
   ```
   subfme. 6,4
   
   # Assume GPR 4 contains 0x8004 3000.
   # Assume the Carry bit is set to zero.
   # GPR 6 now contains 0x4F8E CFFE.
   ```

3. The following code adds the one’s complement of the contents of GPR 4, the Carry bit of the Fixed-Point Exception Register, and x’FFFFFFF’, stores the result in GPR 6, and sets the Fixed-Point Exception Register to reflect the result of the operation:
   
   ```
   subfmeo 6,4
   
   # Assume GPR 4 contains 0xEF00 FFFF.
   # Assume the Carry bit is set to one.
   # GPR 6 now contains 0x1000 0000.
   ```

4. The following code adds the one’s complement of the contents of GPR 4, the Carry bit of the Fixed-Point Exception Register, and x’FFFFFFF’, stores the result in GPR 6, and sets Condition Register Field 0 and the Fixed-Point Exception Register to reflect the result of the operation:
   
   ```
   subfmeo. 6,4
   
   # Assume GPR 4 contains 0xEF00 FFFF.
   # Assume the Carry bit is set to zero.
   # GPR 6 now contains 0x0FFF FFFF.
   ```

### Related Information

- [Fixed-Point Processor](#)
- [Fixed-Point Arithmetic Instructions](#)
subfze or sfze (Subtract from Zero Extended) Instruction

Purpose
Adds the one’s complement of the contents of a general-purpose register, the Carry bit in the Fixed-Point Exception Register, and 0 and places the result in a second general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RT</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>//</td>
</tr>
<tr>
<td>21</td>
<td>OE</td>
</tr>
<tr>
<td>22-30</td>
<td>200</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

PowerPC

<table>
<thead>
<tr>
<th>Syntax</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>subfze</td>
<td>RT RA</td>
</tr>
<tr>
<td>subfze.</td>
<td>RT RA</td>
</tr>
<tr>
<td>subfzeo</td>
<td>RT RA</td>
</tr>
<tr>
<td>subfzeo.</td>
<td>RT RA</td>
</tr>
</tbody>
</table>

POWER family

<table>
<thead>
<tr>
<th>Syntax</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sfze</td>
<td>RT RA</td>
</tr>
<tr>
<td>sfze.</td>
<td>RT RA</td>
</tr>
<tr>
<td>sfzeo</td>
<td>RT RA</td>
</tr>
<tr>
<td>sfzeo.</td>
<td>RT RA</td>
</tr>
</tbody>
</table>

Description
The subfze and sfze instructions add the one’s complement of the contents of general-purpose register (GPR) RA, the Carry bit of the Fixed-Point Exception Register, and x’00000000’ and store the result in the target GPR RT.

The subfze instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

The sfze instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>subfze</td>
<td>0</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>subfze.</td>
<td>0</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>subfzeo</td>
<td>1</td>
<td>SO,OV,CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>subfzeo.</td>
<td>1</td>
<td>SO,OV,CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
<tr>
<td>sfze</td>
<td>0</td>
<td>CA</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>sfze.</td>
<td>0</td>
<td>CA</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>
The four syntax forms of the `subfze` instruction, and the four syntax forms of the `sfze` instruction, always affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

### Parameters

*RT*  
Specifies target general-purpose register where result of operation is stored.  
*RA*  
Specifies source general-purpose register for operation.

### Examples

1. The following code adds the one’s complement of the contents of GPR 4, the Carry bit, and zero and stores the result in GPR 6:  
   # Assume GPR 4 contains 0x9000 3000.  
   # Assume the Carry bit is set to one.  
   subfze 6,4  
   # GPR 6 now contains 0x6FFD 0000.

2. The following code adds the one’s complement of the contents of GPR 4, the Carry bit, and zero, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:  
   # Assume GPR 4 contains 0x8004 3000.  
   # Assume the Carry bit is set to one.  
   subfze 6,4  
   # GPR 6 now contains 0x4FFD 0000.

3. The following code adds the one’s complement of the contents of GPR 4, the Carry bit, and zero, stores the result in GPR 6, and sets the Fixed-Point Exception Register to reflect the result of the operation:  
   # Assume GPR 4 contains 0xEFFF FFFF.  
   # Assume the Carry bit is set to zero.  
   subfzeo 6,4  
   # GPR 6 now contains 0x1000 0000.

4. The following code adds the one’s complement of the contents of GPR 4, the Carry bit, and zero, stores the result in GPR 6, and sets Condition Register Field 0 and the Fixed-Point Exception Register to reflect the result of the operation:  
   # Assume GPR 4 contains 0x70FB 6500.  
   # Assume the Carry bit is set to zero.  
   subfzeo 6,4  
   # GPR 6 now contains 0x8F04 9AFF.

### Related Information

- [Fixed-Point Processor](#)  
- [Fixed-Point Arithmetic Instructions](#)
svc (Supervisor Call) Instruction

Purpose
Generates a supervisor call interrupt.

Note: The svc instruction is supported only in the POWER family architecture.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>17</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-19</td>
<td>FL1</td>
</tr>
<tr>
<td>20-26</td>
<td>LEV</td>
</tr>
<tr>
<td>27-29</td>
<td>FL2</td>
</tr>
<tr>
<td>30</td>
<td>SA</td>
</tr>
<tr>
<td>31</td>
<td>LK</td>
</tr>
</tbody>
</table>

POWER family

svc

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>17</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-29</td>
<td>SV</td>
</tr>
<tr>
<td>30</td>
<td>SA</td>
</tr>
<tr>
<td>31</td>
<td>LK</td>
</tr>
</tbody>
</table>

svcl

svca

svcla

Description
The svc instruction generates a supervisor call interrupt and places bits 16-31 of the svc instruction into bits 0-15 of the Count Register (CR) and bits 16-31 of the Machine State Register (MSR) into bits 16-31 of the CR.

Consider the following when using the svc instruction:

- If the SVC Absolute bit (SA) is set to 0, the instruction fetch and execution continues at one of the 128 offsets, b'1''ll LEV llb'000000', to the base effective address (EA) indicated by the setting of the IP bit of the MSR. FL1 and FL2 fields could be used for passing data to the SVC routine but are ignored by hardware.
- If the SVC Absolute bit (SA) is set to 1, then instruction fetch and execution continues at the offset, x'1FE0', to the base EA indicated by the setting of the IP bit of the MSR.
• If the Link bit (LK) is set to 1, the EA of the instruction following the `svc` instruction is placed in the Link Register.

**Notes:**
1. To ensure correct operation, an `svc` instruction must be preceded by an unconditional branch or a CR instruction. If a useful instruction cannot be scheduled as specified, use a no-op version of the `cror` instruction with the following syntax:
   
   `cror BT, BA, BB`  
   No-op when `BT = BA = BB`

2. The `svc` instruction has the same op code as the `sc` (System Call) instruction.

The `svc` instruction has four syntax forms. Each syntax form affects the MSR.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Link Bit (LK)</th>
<th>SVC Absolute Bit (SA)</th>
<th>Machine State Register Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>svc</td>
<td>0</td>
<td>0</td>
<td>EE,PR,FE set to zero</td>
</tr>
<tr>
<td>svcl</td>
<td>1</td>
<td>0</td>
<td>EE,PR,FE set to zero</td>
</tr>
<tr>
<td>svca</td>
<td>0</td>
<td>1</td>
<td>EE,PR,FE set to zero</td>
</tr>
<tr>
<td>svcla</td>
<td>1</td>
<td>1</td>
<td>EE,PR,FE set to zero</td>
</tr>
</tbody>
</table>

The four syntax forms of the `svc` instruction never affect the FP, ME, AL, IP, IR, or DR bits of the MSR. The EE, PR, and FE bits of the MSR are always set to 0. The Fixed-Point Exception Register and Condition Register Field 0 are unaffected by the `svc` instruction.

**Parameters**

- **LEV** Specifies execution address.
- **FL1** Specifies field for optional data passing to SVC routine.
- **FL2** Specifies field for optional data passing to SVC routine.
- **SV** Specifies field for optional data passing to SVC routine.

**Related Information**
The `cror` (Condition Register OR) instruction, `sc` (System Call) instruction.

- Branch Processor
- System Call Instructions
- Functional Differences for POWER family and PowerPC Instructions

**sync (Synchronize) or dcs (Data Cache Synchronize) Instruction**

**Purpose**
The PowerPC instruction, `sync`, ensures that all previous instructions have completed before the next instruction is initiated.

The POWER family instruction, `dcs`, causes the processor to wait until all data cache lines have been written.
Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-9</td>
<td>///</td>
</tr>
<tr>
<td>10</td>
<td>L</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>598</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

PowerPC
sync

POWER family
dcs

Description
The PowerPC instruction, sync, provides an ordering function that ensures that all instructions initiated prior to the sync instruction complete, and that no subsequent instructions initiate until after the sync instruction completes. When the sync instruction completes, all storage accesses initiated prior to the sync instruction are complete.

The L field is used to specify a heavyweight sync (L = 0) or a lightweight sync (L = 1).

Note: The sync instruction takes a significant amount of time to complete. The eleio (Enforce In-order Execution of I/O) instruction is more appropriate for cases where the only requirement is to control the order of storage references to I/O devices.

The POWER family instruction, dcs, causes the processor to wait until all data cache lines being written or scheduled for writing to main memory have finished writing.

The dcs and sync instructions have one syntax form and do not affect the Fixed-Point Exception Register. If the Record (Rc) bit is set to 1, the instruction form is invalid.

Parameters
L    Specifies heavyweight or a lightweight sync.

Examples
The following code makes the processor wait until the result of the dcbf instruction is written into main memory:

```assembly
# Assume that GPR 4 holds 0x0000 3000.
dcbf 1,4
sync
# Wait for memory to be updated.
```
Related Information
"eieio (Enforce In-Order Execution of I/O) Instruction" on page 196.
Chapter 2, “Processing and Storage,” on page 11.

td (Trap Double Word) Instruction

Purpose
Generate a program interrupt when a specific condition is true.

This instruction should only be used on 64-bit PowerPC processors running a 64-bit application.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>TO</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-20</td>
<td>B</td>
</tr>
<tr>
<td>21-30</td>
<td>68</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

PowerPC64

td TO RA RB

Description
The contents of general-purpose register (GPR) RA are compared with the contents of GPR RB. If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then a trap-type program interrupt is generated.

The TO bit conditions are defined as follows:

<table>
<thead>
<tr>
<th>TO bit</th>
<th>ANDed with Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Compares Less Than.</td>
</tr>
<tr>
<td>1</td>
<td>Compares Greater Than.</td>
</tr>
<tr>
<td>2</td>
<td>Compares Equal.</td>
</tr>
<tr>
<td>3</td>
<td>Compares Logically Less Than.</td>
</tr>
<tr>
<td>4</td>
<td>Compares Logically Greater Than.</td>
</tr>
</tbody>
</table>

Parameters

TO Specifies TO bits that are ANDed with compare results.
RA Specifies source general-purpose register for compare.
RB Specifies source general-purpose register for compare.

Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.
Examples
The following code generates a program interrupt:

```plaintext
# Assume GPR 3 holds 0x0000_0000_0000_0001.
# Assume GPR 4 holds 0x0000_0000_0000_0000.
td 0x2,3,4 # A trap type Program Interrupt occurs.
```

Related Information
- [Branch Processor](#)
- [Fixed-Point Trap Instructions](#)

**tdi (Trap Double Word Immediate) Instruction**

**Purpose**
Generate a program interrupt when a specific condition is true.

This instruction should only be used on 64-bit PowerPC processors running a 64-bit application.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>02</td>
</tr>
<tr>
<td>6-10</td>
<td>TO</td>
</tr>
<tr>
<td>11-15</td>
<td>A</td>
</tr>
<tr>
<td>16-31</td>
<td>SIMM</td>
</tr>
</tbody>
</table>

**Description**
The contents of general-purpose register RA are compared with the sign-extended value of the SIMM field. If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

The TO bit conditions are defined as follows:

<table>
<thead>
<tr>
<th>TO bit</th>
<th>ANDed with Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Compares Less Than.</td>
</tr>
<tr>
<td>1</td>
<td>Compares Greater Than.</td>
</tr>
<tr>
<td>2</td>
<td>Compares Equal.</td>
</tr>
<tr>
<td>3</td>
<td>Compares Logically Less Than.</td>
</tr>
<tr>
<td>4</td>
<td>Compares Logically Greater Than.</td>
</tr>
</tbody>
</table>

**Parameters**
- **TO** Specifies TO bits that are ANDed with compare results.
- **RA** Specifies source general-purpose register for compare.
- **SIMM** 16-bit two’s-complement value which will be sign-extended for comparison.
Implementation
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Related Information
Branch Processor

Fixed-Point Trap Instructions

**tlbie or tlbi (Translation Look-Aside Buffer Invalidate Entry) Instruction**

**Purpose**
Makes a translation look-aside buffer entry invalid for subsequent address translations.

**Notes:**
1. The tlbie instruction is optional for the PowerPC architecture. It is supported on PowerPC 601 RISC Microprocessor, PowerPC 603 RISC Microprocessor and PowerPC 604 RISC Microprocessor.
2. tlbi is a POWER family instruction.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-9</td>
<td>///</td>
</tr>
<tr>
<td>10</td>
<td>L</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>306</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

**PowerPC**

`tlbie` "RB" on page 452  "L" on page 452

**POWER family**

`tlbi` "RA" on page 452  "RB" on page 452

**Description**
The PowerPC instruction **tlbie** searches the Translation Look-Aside Buffer (TLB) for an entry corresponding to the effective address (EA). The search is done regardless of the setting of Machine State Register (MSR) Instruction Relocate bit or the MSR Data Relocate bit. The search uses a portion of the EA including the least significant bits, and ignores the content of the Segment Registers. Entries that satisfy the search criteria are made invalid so will not be used to translate subsequent storage accesses.

The POWER family instruction **tlbi** expands the EA to its virtual address and invalidates any information in the TLB for the virtual address, regardless of the setting of MSR Instruction Relocate bit or the MSR Data Relocate bit. The EA is placed into the general-purpose register (GPR) RA.  

Consider the following when using the POWER family instruction **tlbi**:
If GPR RA is not 0, the EA is the sum of the contents of GPR RA and GPR RB. If GPR RA is 0, EA is the sum of the contents of GPR RB and 0.

- If GPR RA is not 0, EA is placed into GPR RA.
- If EA specifies an I/O address, the instruction is treated as a no-op, but if GPR RA is not 0, EA is placed into GPR RA.

The L field is used to specify a 4 KB page size (L = 0) or a large page size (L = 1).

The tlbie and tlbi instructions have one syntax form and do not affect the Fixed-Point Exception Register. If the Record bit (Rc) is set to 1, the instruction form is invalid.

**Parameters**

The following parameter pertains to the PowerPC instruction, *tlbie*, only:

- RB  Specifies the source general-purpose register containing the EA for the search.
- L   Specifies the page size.

The following parameters pertain to the POWER family instruction, *tlbi*, only:

- RA  Specifies the source general-purpose register for EA calculation and, if RA is not GPR 0, the target general-purpose register for operation.
- RB  Specifies source general-purpose register for EA calculation.

**Security**

The *tlbie* and *tlbi* instructions are privileged.

**Related Information**

Chapter 2, “Processing and Storage,” on page 11.

**tlblld (Load Data TLB Entry) Instruction**

**Purpose**

Loads the data Translation Look-Aside Buffer (TLB) entry to assist a TLB reload function performed in software on the PowerPC 603 RISC Microprocessor.

**Notes:**

1. The *tlblld* instruction is supported only on the PowerPC 603 RISC Microprocessor. It is not part of the PowerPC architecture and not part of the POWER family architecture.
2. TLB reload is usually done by the hardware, but on the PowerPC 603 RISC Microprocessor this is done by software.
3. When AIX is installed on a system using the PowerPC 603 RISC Microprocessor, software to perform the TLB reload function is provided as part of the operating system. You are likely to need to use this instruction only if you are writing software for the PowerPC 603 RISC Microprocessor intended to operate without AIX.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
</tbody>
</table>
### Description
For better understanding, the following information is presented:
- Information about a typical TLB reload function that would call the `tlbld` instruction.
- An explanation of what the `tlbld` instruction does.

#### Typical TLB Reload Function
In the processing of the address translation, the Effective Address (EA) is first translated into a Virtual Address (VA). The part of the Virtual Address is used to select the TLB entry. If an entry is not found in the TLB, a miss is detected. When a miss is detected, the EA is loaded into the data TLB Miss Address (DMISS) register. The first word of the target Page Table Entry is loaded into the data TLB Miss Compare (DCMP) register. A routine is invoked to compare the content of DCMP with all the entries in the primary Page Table Entry Group (PTEG) pointed to by the HASH1 register and all the entries in the secondary PTEG pointed to by the HASH2 register. When there is a match, the `tlbld` instruction is invoked.

#### tlbld Instruction Function
The `tlbld` instruction loads the data Translation Look-Aside Buffer (TLB) entry selected by the content of register `RB` in the following way:
- The content of the data TLB Miss Compare (DCMP) register is loaded into the higher word of the data TLB entry.
- The contents of the RPA register and the data TLB Miss Address (DMISS) register are merged and loaded into the lower word of the data TLB entry.

The `tlbld` instruction has one syntax form and does not affect the Fixed-Point Exception Register. If the Record bit (Rc) is set to 1, the instruction form is invalid.

#### Parameters
- **RB** Specifies the source general-purpose register for EA.

#### Security
The `tlbld` instruction is privileged.

#### Related Information
- "tlbli (Load Instruction TLB Entry) Instruction" on page 454.

*PowerPC 603 RISC Microprocessor User’s Manual.*
tlbli (Load Instruction TLB Entry) Instruction

Purpose
Loads the instruction Translation Look-Aside Buffer (TLB) entry to assist a TLB reload function performed in software on the PowerPC 603 RISC Microprocessor.

Notes:
1. The tlbli instruction is supported only on the PowerPC 603 RISC Microprocessor. It is not part of the PowerPC architecture and not part of the POWER family architecture.
2. TLB reload is usually done by the hardware, but on the PowerPC 603 RISC Microprocessor this is done by software.
3. When AIX is installed on a system using the PowerPC 603 RISC Microprocessor, software to perform the TLB reload function is provided as part of the operating system. You are likely to need to use this instruction only if you are writing software for the PowerPC 603 RISC Microprocessor intended to operate without AIX.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>1010</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

PowerPC 603 RISC Microprocessor

tlbli  

Description
For better understanding, the following information is presented:
• Information about a typical TLB reload function that would call the tlbli instruction.
• An explanation of what the tlbli instruction does.

Typical TLB Reload Function
In the processing of the address translation, the Effective Address (EA) is first translated into a Virtual Address (VA). The part of the Virtual Address is used to select the TLB entry. If an entry is not found in the TLB, a miss is detected. When a miss is detected, the EA is loaded into the instruction TLB Miss Address (IMISS) register. The first word of the target Page Table Entry is loaded into the instruction TLB Miss Compare (ICMP) register. A routine is invoked to compare the content of ICMP with all the entries in the primary Page Table Entry Group (PTEG) pointed to by the HASH1 register and with all the entries in the secondary PTEG pointed to by the HASH2 register. When there is a match, the tlbli instruction is invoked.

tlbli Instruction Function
The tlbli instruction loads the instruction Translation Look-Aside Buffer (TLB) entry selected by the content of register RB in the following way:
• The content of the instruction TLB Miss Compare (DCMP) register is loaded into the higher word of the instruction TLB entry.
• The contents of the RPA register and the instruction TLB Miss Address (IMISS) register are merged and loaded into the lower word of the instruction TLB entry.
The `tlbli` instruction has one syntax form and does not affect the Fixed-Point Exception Register. If the Record bit (Rc) is set to 1, the instruction form is invalid.

**Parameters**

*RB*  Specifies the source general-purpose register for EA.

**Security**

The `tlbli` instruction is privileged.

**Related Information**

“`tlbld (Load Data TLB Entry) Instruction` on page 452.

*PowerPC 603 RISC Microprocessor User's Manual.*

---

**tlbsync (Translation Look-Aside Buffer Synchronize) Instruction**

**Purpose**

Ensures that a `tlbie` and `tlbia` instruction executed by one processor has completed on all other processors.

*Note:* The `tlbsync` instruction is defined only in the PowerPC architecture and is an optional instruction. It is supported on the PowerPC 603 RISC Microprocessor and on the PowerPC 604 RISC Microprocessor, but not on the PowerPC 601 RISC Microprocessor.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>///</td>
</tr>
<tr>
<td>11-15</td>
<td>///</td>
</tr>
<tr>
<td>16-20</td>
<td>///</td>
</tr>
<tr>
<td>21-30</td>
<td>566</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

*PowerPC*

**tlbsync**

**Description**

The `tlbsync` instruction does not complete until all previous `tlbie` and `tlbia` instructions executed by the processor executing the `tlbsync` instruction have been received and completed by all other processors.

The `tlbsync` instruction has one syntax form and does not affect the Fixed-Point Exception Register. If the Record bit (Rc) is set to 1, the instruction form is invalid.

**Security**

The `tlbsync` instruction is privileged.
Related Information

Processing and Storage

**tw or t (Trap Word) Instruction**

**Purpose**
Generates a program interrupt when a specified condition is true.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>TO</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>4</td>
</tr>
<tr>
<td>31</td>
<td>/</td>
</tr>
</tbody>
</table>

PowerPC

**tw**

```
TO, RA, RB
```

POWER family

**t**

```
TO, RA, RB
```

See [Extended Mnemonics of Fixed-Point Trap Instructions](#) for more information.

**Description**
The **tw** and **t** instructions compare the contents of general-purpose register (GPR) **RA** with the contents of GPR **RB**, AND the compared results with **TO**, and generate a trap-type Program Interrupt if the result is not 0.

The **TO** bit conditions are defined as follows.

<table>
<thead>
<tr>
<th>TO bit</th>
<th>ANDed with Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Compares Less Than.</td>
</tr>
<tr>
<td>1</td>
<td>Compares Greater Than.</td>
</tr>
<tr>
<td>2</td>
<td>Compares Equal.</td>
</tr>
<tr>
<td>3</td>
<td>Compares Logically Less Than.</td>
</tr>
<tr>
<td>4</td>
<td>Compares Logically Greater Than.</td>
</tr>
</tbody>
</table>

The **tw** and **t** instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

**Parameters**

- **TO** Specifies **TO** bits that are ANDed with compare results.
- **RA** Specifies source general-purpose register for compare.
- **RB** Specifies source general-purpose register for compare.
Examples
The following code generates a Program Interrupt:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 7 contains 0x789A 789B.
tw 0x10,4,7
# A trap type Program Interrupt occurs.
```

Related Information
- [Branch Processor](#)
- [Fixed-Point Trap Instructions](#)

**twi or ti (Trap Word Immediate) Instruction**

**Purpose**
Generates a program interrupt when a specified condition is true.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>03</td>
</tr>
<tr>
<td>6-10</td>
<td>TO</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>SI</td>
</tr>
</tbody>
</table>

**PowerPC**

twi 

**POWER family**
ti

See [Extended Mnemonics of Fixed-Point Trap Instructions](#) for more information.

**Description**
The `twi` and `ti` instructions compare the contents of general-purpose register (GPR) RA with the sign extended SI field, AND the compared results with TO, and generate a trap-type program interrupt if the result is not 0.

The TO bit conditions are defined as follows.

<table>
<thead>
<tr>
<th>TO bit</th>
<th>ANDed with Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Compares Less Than.</td>
</tr>
<tr>
<td>1</td>
<td>Compares Greater Than.</td>
</tr>
<tr>
<td>2</td>
<td>Compares Equal.</td>
</tr>
<tr>
<td>3</td>
<td>Compares Logically Less Than.</td>
</tr>
<tr>
<td>4</td>
<td>Compares Logically Greater Than.</td>
</tr>
</tbody>
</table>

The `twi` and `ti` instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.
Parameters

TO  Specifies TO bits that are ANDed with compare results.
RA  Specifies source general-purpose register for compare.
SI  Specifies sign-extended value for compare.

Examples

The following code generates a Program Interrupt:

```
# Assume GPR 4 holds 0x0000 0010.
twi 0x4,4,0x10
# A trap type Program Interrupt occurs.
```

Related Information

- [Branch Processor](#)
- [Fixed-Point Trap Instructions](#)

xor (XOR) Instruction

Purpose

XORs the contents of two general-purpose registers and places the result in another general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>31</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-20</td>
<td>RB</td>
</tr>
<tr>
<td>21-30</td>
<td>316</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

```
xor  RA RS RB
xor. RA RS RB
```

Description

The xor instruction XORs the contents of general-purpose register (GPR) RS with the contents of GPR RB and stores the result in GPR RA.

The xor instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

<table>
<thead>
<tr>
<th>Syntax Form</th>
<th>Overflow Exception (OE)</th>
<th>Fixed-Point Exception Register</th>
<th>Record Bit (Rc)</th>
<th>Condition Register Field 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>xor.</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>LT,GT,EQ,SO</td>
</tr>
</tbody>
</table>
The two syntax forms of the \texttt{xor} instruction never affect the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

**Parameters**

- \textit{RA} Specifies target general-purpose register where result of operation is stored.
- \textit{RS} Specifies source general-purpose register for operation.
- \textit{RB} Specifies source general-purpose register for operation.

**Examples**

1. The following code XORs the contents of GPR 4 and GPR 7 and stores the result in GPR 6:
   
   ```
   # Assume GPR 4 contains 0x9000 3000.
   # Assume GPR 7 contains 0x789A 789B.
   xor 6,4,7
   # GPR 6 now contains 0xE89A 489B.
   ```

2. The following code XORs the contents of GPR 4 and GPR 7, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:
   
   ```
   # Assume GPR 4 contains 0xB004 3000.
   # Assume GPR 7 contains 0x789A 789B.
   xor 6,4,7
   # GPR 6 now contains 0xC89E 489B.
   ```

**Related Information**

- Fixed-Point Processor
- Fixed-Point Logical Instructions

---

\textbf{xori or xoril (XOR Immediate) Instruction}

**Purpose**

XORs the lower 16 bits of a general-purpose register with a 16-bit unsigned integer and places the result in another general-purpose register.

**Syntax**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>26</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>UI</td>
</tr>
</tbody>
</table>

\begin{Verbatim}
\textbf{PowerPC}
\textbf{xori} \quad \textbf{RA} \textbf{RS} \textbf{UI}
\end{Verbatim}

\begin{Verbatim}
\textbf{POWER family}
\textbf{xoril} \quad \textbf{RA} \textbf{RS} \textbf{UI}
\end{Verbatim}
Description
The xori and xoril instructions XOR the contents of general-purpose register (GPR) RS with the concatenation of x’0000’ and a 16-bit unsigned integer UI and store the result in GPR RA.

The xori and xoril instructions have only one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters
RA Specifies target general-purpose register where result of operation is stored.
RS Specifies source general-purpose register for operation.
UI Specifies 16-bit unsigned integer for operation.

Examples
The following code XORs GPR 4 with 0x0000 5730 and places the result in GPR 6:
# Assume GPR 4 contains 0x7B41 92C0.
xori 6,4,0x5730
# GPR 6 now contains 0x7B41 C5F0.

Related Information
Fixed-Point Processor.
Fixed-Point Logical Instructions.

xoris or xoriu (XOR Immediate Shift) Instruction

Purpose
XORs the upper 16 bits of a general-purpose register with a 16-bit unsigned integer and places the result in another general-purpose register.

Syntax

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>27</td>
</tr>
<tr>
<td>6-10</td>
<td>RS</td>
</tr>
<tr>
<td>11-15</td>
<td>RA</td>
</tr>
<tr>
<td>16-31</td>
<td>UI</td>
</tr>
</tbody>
</table>

PowerPC
xoris  RA RS UI

POWER family
xoriu RA RS UI

Description
The xoris and xoriu instructions XOR the contents of general-purpose register (GPR) RS with the concatenation of a 16-bit unsigned integer UI and 0x’0000’ and store the result in GPR RA.
The xoris and xoriu instructions have only one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

RA  Specifies target general-purpose register where result of operation is stored.
RS  Specifies source general-purpose register for operation.
UI  Specifies 16-bit unsigned integer for operation.

Example

The following code XORs GPR 4 with 0x0079 0000 and stores the result in GPR 6:

```
# Assume GPR 4 holds 0x9000 3000.
xoris 6,4,0x0079
# GPR 6 now holds 0x9079 3000.
```

Related Information

[Fixed-Point Processor](#).

[Fixed-Point Logical Instructions](#).
Chapter 9. Pseudo-ops

This chapter provides an overview of assembler pseudo-ops and reference information for all pseudo-ops.

Pseudo-ops Overview

A pseudo-operation, commonly called a pseudo-op, is an instruction to the assembler that does not generate any machine code. The assembler resolves pseudo-ops during assembly, unlike machine instructions, which are resolved only at runtime. Pseudo-ops are sometimes called assembler instructions, assembler operators, or assembler directives.

In general, pseudo-ops give the assembler information about data alignment, block and segment definition, and base register assignment. The assembler also supports pseudo-ops that give the assembler information about floating point constants and symbolic debugger information (dbx).

While they do not generate machine code, the following pseudo-ops can change the contents of the assembler’s location counter:

- .align
- .byte
- .comm
- .csect
- .double
- .dsect
- .float
- .lcomm
- .long
- .org
- .short
- .space
- .string
- .vbyte

Pseudo-ops Grouped by Function

Pseudo-ops can be related according to functionality into the following groups:

- “Data Alignment” on page 464
- “Data Definition” on page 464
- “Storage Definition” on page 464
- “Addressing” on page 464
- “Assembler Section Definition” on page 464
- “External Symbol Definition” on page 464
- “Static Symbol Definition” on page 465
- “Support for Calling Conventions” on page 465
- “Miscellaneous” on page 465
- “Symbol Table Entries for Debuggers” on page 465
- “Target Environment Indication” on page 465

© Copyright IBM Corp. 1997, 2006
Data Alignment

The following pseudo-op is used in the data or text section of a program:

- `.align`

Data Definition

The following pseudo-ops are used for data definition:

- `.byte`
- `.double`
- `.float`
- `.long`
- `.short`
- `.string`
- `.vbyte`

In most instances, use these pseudo-ops to create data areas to be used by a program, as shown by this example.

```assembly
.csect data[rw]
greeting: .long 'H','O','W','D','Y
```

Storage Definition

The following pseudo-ops define or map storage:

- `.dsect`
- `.space`

Addressing

The following pseudo-ops assign or dismiss a register as a base register:

- `.drop`
- `.using`

Assembler Section Definition

The following pseudo-ops define the sections of an assembly language program:

- `.comm`
- `.csect`
- `.lcomm`
- `.tc`
- `.toc`

External Symbol Definition

The following pseudo-ops define a variable as a global variable or an external variable (variables defined in external modules):

- `.extern`
Static Symbol Definition
The following pseudo-op defines a static symbol:
  • `.lglobl`

Support for Calling Conventions
The following pseudo-op defines a debug traceback tag for performing tracebacks when debugging programs:
  • `.tbtag`

Miscellaneous
The following pseudo-ops perform miscellaneous functions:

  • `.hash` Provides type-checking information.
  • `.org` Sets the value of the current location counter.
  • `.ref` Creates a special type entry in the relocation table.
  • `.rename` Creates a synonym or alias for an illegal or undesirable name.
  • `.set` Assigns a value and type to a symbol.
  • `.source` Identifies the source language type.
  • `.tocof` Defines a symbol as the table of contents (TOC) of another module.
  • `.xline` Provides file and line number information.

Symbol Table Entries for Debuggers
The following pseudo-ops provide additional information which is required by the symbolic debugger (`dbx`):
  • `.bb`
  • `.bc`
  • `.bf`
  • `.bi`
  • `.bs`
  • `.eb`
  • `.ec`
  • `.ef`
  • `.ei`
  • `.es`
  • `.file`
  • `.function`
  • `.line`
  • `.stabx`
  • `.xline`

Target Environment Indication
The following pseudo-op defines the intended target environment:
  • `.machine`
Notational Conventions

White space is required unless otherwise specified. A space may optionally occur after a comma. White space may consist of one or more white spaces.

Some pseudo-ops may not use labels. However, with the exception of the .csect pseudo-op, you can put a label in front of a pseudo-op statement just as you would for a machine instruction statement.

The following notational conventions are used to describe pseudo-ops:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Any valid label.</td>
</tr>
<tr>
<td>Register</td>
<td>A general-purpose register.</td>
</tr>
<tr>
<td>Number</td>
<td>An expression that evaluates to an integer.</td>
</tr>
<tr>
<td>Expression</td>
<td>Unless otherwise noted, the variable signifies a relocatable constant or absolute expression.</td>
</tr>
<tr>
<td>FloatingConstant</td>
<td>A floating-point constant.</td>
</tr>
<tr>
<td>StringConstant</td>
<td>A string constant.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Brackets enclose optional operands except in the .csect and .tc Pseudo-op, which require brackets in syntax.</td>
</tr>
</tbody>
</table>

.align Pseudo-op

Purpose
Advances the current location counter until a boundary specified by the Number parameter is reached.

Syntax

```
.align "Number"
```

Description
The .align pseudo-op is normally used in a control section (csect) that contains data.

If the Number parameter evaluates to 0, alignment occurs on a byte boundary. If the Number parameter evaluates to 1, alignment occurs on a halfword boundary. If the Number parameter evaluates to 2, alignment occurs on a word boundary. If the Number parameter evaluates to 3, alignment occurs on a doubleword boundary.

If the location counter is not aligned as specified by the Number parameter, the assembler advances the current location counter until the number of low-order bits specified by the Number parameter are filled with the value 0 (zero).

If the .align pseudo-op is used within a .csect pseudo-op of type PR or GL which indicates a section containing instructions, alignment occurs by padding with nop (no-operation) instructions. In this case, the no-operation instruction is equivalent to a branch to the following instruction. If the align amount is less than a fullword, the padding consists of zeros.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Specifies an absolute expression that evaluates to an integer value from 0 to 12, inclusive. The value indicates the log base 2 of the desired alignment. For example, an alignment of 8 (a doubleword) would be represented by an integer value of 3; an alignment of 4096 (one page) would be represented by an integer value of 12.</td>
</tr>
</tbody>
</table>
Examples
The following example demonstrates the use of the .align pseudo-op:

```assembly
.csect progdata[RW]
.byte 1
    # Location counter now at odd number
.align 1
    # Location counter is now at the next
    # halfword boundary.
.byte 3,4

.align 2  # Insure that the label cont
    # and the .long pseudo-op are
    # aligned on a full word
    # boundary.
cont: .long 5004381
```

Related Information
“Pseudo-ops Overview” on page 463.
“.byte Pseudo-op” on page 470, “.comm Pseudo-op” on page 471, “.csect Pseudo-op” on page 473,
“.double Pseudo-op” on page 475, “.float Pseudo-op” on page 483, “.long Pseudo-op” on page 489, “.short
Pseudo-op” on page 497.

.bb Pseudo-op

Purpose
Identifies the beginning of an inner block and provides information specific to the beginning of an inner
block.

Syntax
```assembly
.bb "Number"
```

Description
The .bb pseudo-op provides symbol table information necessary when using the symbolic debugger.

The .bb pseudo-op has no other effect on assembly and is customarily inserted by a compiler.

Parameters

*Number* Specifies the line number in the original source file on which the inner block begins.

Examples
The following example demonstrates the use of the .bb pseudo-op:

```assembly
.bb 5
```
.bc Pseudo-op

Purpose
Identifies the beginning of a common block and provides information specific to the beginning of a common block.

Syntax
```
.bc StringConstant
```

Description
The .bc pseudo-op provides symbol table information necessary when using the symbolic debugger.

The .bc pseudo-op has no other effect on assembly and is customarily inserted by a compiler.

Parameters

- `StringConstant` Represents the symbol name of the common block as defined in the original source file.

Examples
The following example demonstrates the use of the .bc pseudo-op:
```
.bc "commonblock"
```

Related Information

- Pseudo-ops Overview

The .ec pseudo-op.

.bc Pseudo-op

Purpose
Identifies the beginning of a function and provides information specific to the beginning of a function.

Syntax
```
 bf Number
```

Description
The .bf pseudo-op provides symbol table information necessary when using the symbolic debugger.

The .bf pseudo-op has no other effect on assembly and is customarily inserted by a compiler.

Note: The .function pseudo-op must be used if the .bf pseudo-op is used.
Parameters

*Number* Represents the absolute line number in the original source file on which the function begins.

Examples

The following example demonstrates the use of the `.bf` pseudo-op:

```
.bf  5
```

Related Information

Pseudo-ops Overview

The `[ef]` pseudo-op, `function` pseudo-op.

---

**.bi Pseudo-op**

Purpose

Identifies the beginning of an included file and provides information specific to the beginning of an included file.

Syntax

```
.bi [StringConstant]
```

Description

The `.bi` pseudo-op provides symbol table information necessary when using the symbolic debugger.

The `.bi` pseudo-op has no other effect on assembly and is customarily inserted by a compiler.

The `.bi` pseudo-op should be used with the `.line` pseudo-op.

Parameters

*StringConstant* Represents the name of the original source file.

Examples

The following example demonstrates the use of the `.bi` pseudo-op:

```
.bi "file.s"
```

Related Information

Pseudo-ops Overview

The `[ei]` pseudo-op, `line` pseudo-op.

---

**.bs Pseudo-op**

Purpose

Identifies the beginning of a static block and provides information specific to the beginning of a static block.
Syntax

.bs [Name]

Description
The .bs pseudo-op provides symbol table information necessary when using the symbolic debugger.

The .bs pseudo-op has no other effect on assembly and is customarily inserted by a compiler.

Parameters
Name Represents the symbol name of the static block as defined in the original source file.

Examples
The following example demonstrates the use of the .bs pseudo-op:
.lcomm cgdat, 0x2b4
.csect .text[PR]
.bs cgdat
.stabx "ONE:1=Ci2,0,4;",0x254,133,0
.stabx "TWO:S2=G5TWO1:3=Cc5,0,5;,0,40;;",0x258,133,8
.es

Related Information
Pseudo-ops Overview

The .comm pseudo-op, .es pseudo-op, .lcomm pseudo-op.

.byte Pseudo-op

Purpose
Assembles specified values represented by the Expression parameter into consecutive bytes.

Syntax

.byte [Expression,Expression...]

Description
The .byte pseudo-op changes an expression or a string of expressions into consecutive bytes of data. ASCII character constants (for example, 'X') and string constants (for example, Hello, world) can also be assembled using the .byte pseudo-op. Each letter will be assembled into consecutive bytes. However, an expression cannot contain externally defined symbols. Also, an expression value longer than one byte will be truncated on the left.

Parameters

Expression Specifies a value that is assembled into consecutive bytes.

Examples
The following example demonstrates the use of the .byte pseudo-op:
.set olddata,0xCC
.csect data[rw]

mine: .byte 0x3F,0x7+0xA,olddata,0xFF

# Load GPR 3 with the address of csect data[rw].
.csect text[pr]
  1 3,mine(4)

# GPR 3 now holds 0x3F11 CCFF.
# Character constants can be represented in
# several ways:
   .csect data[rw]
   .byte "Hello, world"
   .byte 'H','e','l','l','o',' ','w','o','r','l','d'

# Both of the .byte statements will produce
# 0x4865 6C6C 6F2C 2077 6F72 6C64.

Related Information
Pseudo-ops Overview

The .string pseudo-op, .vbyte pseudo-op.

.comm Pseudo-op

Purpose
Defines an uninitialized block of storage called a common block, which can be common to more than one
module.

Syntax
.comm [Qualname, Expression, Number]

where QualName = Name[[StorageMappingClass]]

Note: Name is required. StorageMappingClass is optional and enclosed within brackets if specified.
RW is the assumed default if StorageMappingClass is omitted.

Description
The .comm pseudo-op defines a block of storage specified by the Qualname parameter. The the block
size is specified in bytes by the Expression parameter.

Note: By convention, use of the TD storage mapping class is restricted to common blocks no more
than four (4) bytes long.

The valid values for StorageMappingClass are RW, TD, UC, and BS. These values are explained in the
article on the .csect pseudo-op. If any other value is used for StorageMappingClass, the default value RW
is used and a warning message is reported if the -w flag is in effect.

If TD is used for the storage mapping class, a block of zeroes, the length specified by the Expression
parameter, will be written into the TOC area as an initialized csect in the .data section. If RW, UC, or BS is
used as the storage mapping class, the block is not initialized in the current module and has symbol type
CM (Common). At load time, the space for CM control sections with RW, UC, or BC storage mapping
classes is created in the .bss section at the end of the .data section.

Several modules can share the same common block. If any of those modules have an external Control
Section (csect) with the same name and the csect with the same name has a storage mapping class other
than BS or UC, then the common block is initialized and becomes that other Control Section. If the
common block has TD as its storage mapping class, the csect will be in the TOC area. This is
accomplished at bind time.

If more than one uninitialized common block with the same Qualname is found at bind time, space is
reserved for the largest one.

A common block can be aligned by using the Number parameter, which is specified as the log base 2 of
the alignment desired.

**Parameters**

*Qualname* Specifies the name and storage mapping class of the common block. If the StorageMappingClass
part of the parameter is omitted, the default value RW is used. Valid StorageMappingClass
values for a common block are RW, TD, UC and BS.

*Expression* Specifies the absolute expression that gives the length of the specified common block in bytes.

*Number* Specifies the optional alignment of the specified common block. This is specified as the log base
2 of the alignment desired. For example, an alignment of 8 (or doubleword) would be 3 and an
alignment of 2048 would be 11. This is similar to the argument for the .align pseudo-op.

**Examples**

1. The following example demonstrates the use of the .comm pseudo-op:

```
.comm proc,5120
# proc is an uninitialized common block of
# storage 5120 bytes long which is
# globally visible.

# Assembler SourceFile A contains:
.comm st,1024
# Assembler SourceFile B contains:
.globl st[RW]
.csect st[RW]
.long 1
.long 2

# Using st in the above two programs refers to
# Control Section st in Assembler SourceFile B.
```

2. This example shows how two different modules access the same data:

a. Source code for C module td2.c:

```
/* This C module named td2.c */
extern long t_data;
extern void mod_s();
main()
{
    t_data = 1234;
    mod_s();
    printf("t_data is %d\n", t_data);
}
```

b. Source for assembler module mod2.s:

```
.file "mod2.s"
csect .mod_s[PR]
globl .mod_s[PR]
.set RTOC, 2
l 5, t_data[TD](RTOC) # Now GPR5 contains the
    # t_data value
ai 5,5,14
stu 5, t_data[TD](RTOC)
```

472 Assembler Language Reference
br
.toc
.comm t_data[TD],4 # t_data is a global symbol
c. Instructions for making executable td2 from the C and assembler source:
as -o mod2.o mod2.s
c -o td2 td2.c mod2.o
d. Running td2 will cause the following to be printed:
t_data is 1248

Related Information
Pseudo-ops Overview

The .align pseudo-op, .csect pseudo-op, .globl pseudo-op, .lcomm pseudo-op, .long pseudo-op.

.csect Pseudo-op

Purpose
Groups code or data into a control section (csect) and gives that csect a name, a storage mapping class, and an alignment.

Syntax
.csect [QualifiedName], [Number]

where QulName = [Name][StorageMappingClass]

Note: The boldfaced brackets containing StorageMappingClass are part of the syntax and do not specify an optional parameter.

Description
The following information discusses using the .csect pseudo-op:

• A csect QualName parameter takes the form:
symbol[XX]
OR
symbol{XX}

where either the [ ] (square brackets) or { } (curly brackets) surround a two- or three-character storage mapping class identifier. Both types of brackets produce the same results.

The QualName parameter can be omitted. If it is omitted, the csect is unnamed and the [PR] StorageMappingClass is used. If a QualName is used, the Name parameter is optional and the StorageMappingClass is required. If no Name is specified, the csect is unnamed.

Each control section has a storage mapping class associated with it that is specified in the qualification part of QualName. The storage mapping class determines the object data section, specifically the .text, .data, or .bss section, in which the control section is grouped. The .text section contains read-only data. The .data and .bss sections contain read/write data.

The storage mapping class also indicates what kind of data should be contained within the control section. Many of the storage mapping classes listed have specific implementation and convention details. In general, instructions can be contained within csects of storage mapping class PR. Modifiable data can be contained within csects of storage mapping class RW.

A csect is associated with one of the following storage mapping classes. Storage mapping class
terminers are not case-sensitive. The storage mapping class terminers are listed in groups for the .text,
.text, and .bss object data sections.

.text Section Storage Mapping Classes

PR  Program Code. Identifies the sections that provide executable instructions for the module.
RO  Read-Only Data. Identifies the sections that contain constants that are not modified during execution.
DB  Debug Table. Identifies a class of sections that have the same characteristics as read-only data.
GL  Glue Code. Identifies a section that has the same characteristics as Program Code. This type of section has code to interface with a routine in another module. Part of the interface code requirement is to maintain TOC addressability across the call.
XO  Extended Operation. Identifies a section of code that has no dependency on the TOC (no references through the TOC). It is intended to reside at a fixed address in memory so that it can be the target of a branch to an absolute address.

Note: This storage mapping class should not be used in assembler source programs.

SV  Supervisor Call. Identifies a section of code that is to be treated as a supervisor call.
TB  Traceback Table. Identifies a section that contains data associated with a traceback table.
TI  Traceback Index. Identifies a section that contains data associated with a traceback index.

.data Section Storage Mapping Classes

TC0  TOC Anchor used only by the predefined TOC symbol. Identifies the special symbol TOC. Used only for the TOC anchor.
TC  TOC Entry. Generally indicates a csect that contains addresses of other csects or global symbols. If it contains only one address, the csect is usually four bytes long.
TD  TOC Entry. Identifies a csect that contains scalar data that can be directly accessed from the TOC. For frequently used global symbols, this is an alternative to indirect access through an address pointer csect within the TOC. By convention, TD sections should not be longer than four bytes. Contains initialized data that can be modified during program execution.
UA  Unknown Type. Identifies a section that contains data of an unknown storage mapping class.
RW  Read/Write Data. Identifies a section that contains data that is known to require change during execution.
DS  Descriptor. Identifies a function descriptor. This information is used to describe function pointers in languages such as C and FORTRAN.

.bss Section Storage Mapping Classes

BS  BSS class. Identifies a section that contains uninitialized read/write data.
UC  Unnamed FORTRAN Common. Identifies a section that contains read/write data.

A csect is one of the following symbol types:

ER  External Reference
SD  CSECT Section Definition
LD  Entry Point - Label Definition
CM  Common (BSS)

- All of the control sections with the same QualName value are grouped together, and a section can be continued with a .csect statement having the same QualName. Different csects can have the same name and different storage mapping classes. Therefore, the storage mapping class identifier must be used when referring to a csect name as an operand of other pseudo-ops or instructions. However, for a given name, only one csect can be externalized. If two or more csects with the same name are externalized, a run error may occur, since the linkage editor treats the csects as duplicate symbol definitions and selects only one of them to use.
- A control section is relocated as a body.
- Control sections with no specified name (Name) are identified with their storage mapping class, and there can be an unnamed control section of each storage mapping class. They are specified with a QualName that only has a storage mapping class (for instance, .csect [RW] has a QualName of [RW]).
• If no `.csect` pseudo-op is specified before any instructions appear, then an unnamed Program Code ([PR]) control section is assumed.

• A csect with the BS or UC storage mapping class will have a csect type of CM (Common), which reserves spaces but has no initialized data. All other control sections defined with the `.csect` pseudo-op are of type SD (Section Definition). The `.comm` or `.lcomm` pseudo-ops can also be used to define control sections of type CM. No external label can be defined in a control section of type CM.

• Do not label `.csect` statements. The `.csect` may be referred to by its `QualifiedName`, and labels may be placed on individual elements of the `.csect`.

### Parameters

**Number**
Specifies an absolute expression that evaluates to an integer value from 0 to 31, inclusive. This value indicates the log base 2 of the desired alignment. For example, an alignment of 8 (a doubleword) would be represented by an integer value of 3; an alignment of 2048 would be represented by an integer value of 11. This is similar to the usage of the `Number` parameter for the `.align` pseudo-op. Alignment occurs at the beginning of the csect. Elements of the csect are not individually aligned.

The `Number` parameter is optional. If it is not specified, the default value is 2.

**QualifiedName**
Specifies a `Name` and `StorageMappingClass` for the control section. If `Name` is not given, the csect is identified with its `StorageMappingClass`.

If neither the `Name` nor the `StorageMappingClass` are given, the csect is unnamed and has a storage mapping class of [PR]. If the `Name` is specified, the `StorageMappingClass` must also be specified.

### Examples

The following example defines three csects:

```assembly
# A csect of name proga with Program Code Storage Mapping Class.
.csect proga [PR]
1h 30,0x64(5)

# A csect of name pdata_ with Read-Only Storage Mapping Class.
.csect pdata_[RO]
11: .long 0x7782
12: .byte 'a','b','c','d','e

# An unnamed csect with Read/Write Storage Mapping Class and doubleword alignment.
.csect [RW],3
.float -5
```

### Related Information

**Pseudo-ops Overview**

The `.comm` pseudo-op, `.globl` pseudo-op, `.lcomm` pseudo-op, `.align` pseudo-op.

---

### .double Pseudo-op

**Purpose**

Stores a double floating-point constant at the next fullword location.

**Syntax**

```
.double [FloatingConstant]
```
Parameters

FloatingConstant Specifies a floating-point constant to be assembled.

Examples
The following example demonstrates the use of the .double pseudo-op:

```
.double 3.4
.double -77
.double 134E12
.double 5e300
.double 0.45
```

Related Information
Pseudo-ops Overview

The .float pseudo-op.

.drop Pseudo-op

Purpose
Stops using a specified register as a base register.

Syntax

```
.drop [Number]
```

Description
The .drop pseudo-op stops a program from using the register specified by the Number parameter as a base register in operations. The .drop pseudo-op does not have to precede the .using pseudo-op when changing the base address, and the .drop pseudo-op does not have to appear at the end of a program.

Parameters

Number Specifies an expression that evaluates to an integer from 0 to 31 inclusive.

Examples
The following example demonstrates the use of the .drop pseudo-op:

```
.using _subrA,5
    # Register 5 can now be used for addressing
    # with displacements calculated
    # relative to _subrA.
    # .using does not load GPR 5 with the address
    # of _subrA. The program must contain the
    # appropriate code to ensure this at runtime.

.
.
.drop 5
    # Stop using Register 5.
.using _subrB,5
    # Now the assembler calculates
    # displacements relative to _subrB
```
Related Information

Pseudo-ops Overview

The .using pseudo-op.

.dsect Pseudo-op

Purpose
Identifies the beginning or the continuation of a dummy control section.

Syntax

.dsect Name

Description
The .dsect pseudo-op identifies the beginning or the continuation of a dummy control section. Actual data declared in a dummy control section is ignored; only the location counter is incremented. All labels in a dummy section are considered to be offsets relative to the beginning of the dummy section. A dsect that has the same name as a previous dsect is a continuation of that dummy control section.

The .dsect pseudo-op can declare a data template that can then be used to map out a block of storage. The .using pseudo-op is involved in doing this.

Parameters

Name  Specifies a dummy control section.

Examples

1. The following example demonstrates the use of the .dsect pseudo-op:

   .dsect datal
d1:   .long 0

   # 1 Fullword
   d2:   .short 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   # 10 Halfwords
   d3:   .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   # 15 bytes
   d4:   .align 3
   #Align to a double word.

   .csect main[PR]
   .using data1,7
   l 5,d2

   # This will actually load
   # the contents of the
   # effective address calculated
   # by adding the offset d2 to
   # that in GPR 7 into GPR 5.

2. The following example contains several source programs which together show the use of .dsect and .using pseudo-ops in implicit-based addressing.
   a. Source program foo_pm.s:
b. Source program bar_pm.s:

```
.csect bar_data[RW]
.long 0xbb
.short 30
.short 40
.globl .bar_pm[PR]
.csect .bar_pm[PR]
.extern l1
.using TOC[TC0], 2
l 7, T.bar_data
b l1
br
.toc
T.bar_data: .tc bar_data[TC], bar_data[RW]
```

c. Source program c1_s:

```
.dsect data1
d1: .long 0
d2: .short 0
d3: .short 0
.globl .c1[PR]
.csect .c1[PR]
.globl l1
l1: .using data1, 7
l 5, d1
stu 5, t_data[TD](2)
br # this br is necessary.
    # without it, prog hangs
.toc
.comm t_data[TD],4
```

d. Source for main program mm.c:

```c
extern long t_data;
main()
{
    int sw;
    sw = 2;
    if ( sw == 2 )
    {
        foo_pm();
        printf ( "when sw is 2, t_data is 0x%x\n", t_data );
    }
    sw = 1;
    if ( sw == 1 )
    {
        bar_pm();
        printf ( "when sw is 1, t_data is 0x%x\n", t_data );
    }
}
```

e. Instructions for creating the executable file from the source:

```
as -o foo_pm.o foo_pm.s
as -o bar_pm.o bar_pm.s
as -o c1.o c1.s
cc -o mm.mm.mm c1.o foo_pm.o bar_pm.o c1.o
```
f. The following is printed if \texttt{mm} is executed:

\begin{itemize}
  \item when \texttt{sw} is 2, \texttt{t_data} is 0xaa
  \item when \texttt{sw} is 1, \texttt{t_data} is 0xbb
\end{itemize}

\section*{Related Information}

\subsection*{Pseudo-ops Overview}

The \texttt{.csect} pseudo-op, \texttt{.using} pseudo-op.

\section*{.eb Pseudo-op}

\subsection*{Purpose}

Identifies the end of an inner block and provides additional information specific to the end of an inner block.

\subsection*{Syntax}

\begin{verbatim}
.eb Number
\end{verbatim}

\subsection*{Description}

The \texttt{.eb} pseudo-op identifies the end of an inner block and provides symbol table information necessary when using the symbolic debugger.

The \texttt{.eb} pseudo-op has no other effect on assembly and is customarily inserted by a compiler.

\subsection*{Parameters}

\begin{description}
  \item[Number] Specifies a line number in the original source file on which the inner block ends.
\end{description}

\subsection*{Examples}

The following example demonstrates the use of the \texttt{.eb} pseudo-op:

\begin{verbatim}
.eb 10
\end{verbatim}

\section*{Related Information}

\subsection*{Pseudo-ops Overview}

The \texttt{.bb} pseudo-op.

\section*{.ec Pseudo-op}

\subsection*{Purpose}

Identifies the end of a common block and provides additional information specific to the end of a common block.

\subsection*{Syntax}

\begin{verbatim}
.ec
\end{verbatim}
Description
The .ec pseudo-op identifies the end of a common block and provides symbol table information necessary when using the symbolic debugger.

The .ec pseudo-op has no other effect on assembly and is customarily inserted by a compiler.

Examples
The following example demonstrates the use of the .ec pseudo-op:

```
.bc "commonblock"
.ec
```

Related Information
Pseudo-ops Overview

The .bc pseudo-op.

---

.ef Pseudo-op

Purpose
Identifies the end of a function and provides additional information specific to the end of a function.

Syntax
```
.ef Number
```

Description
The .ef pseudo-op identifies the end of a function and provides symbol table information necessary when using the symbolic debugger.

The .ef pseudo-op has no other effect on assembly and is customarily inserted by a compiler.

Parameters

Number Specifies a line number in the original source file on which the function ends.

Examples
The following example demonstrates the use of the .ef pseudo-op:

```
.ef 10
```

Related Information
Pseudo-ops Overview

The .bf pseudo-op.

---

ei Pseudo-op

Purpose
Identifies the end of an included file and provides additional information specific to the end of an included file.
Syntax
.es

Description
The .es pseudo-op identifies the end of a static block and provides additional information specific to the end of a static block.

The .es pseudo-op has no other effect on assembly and is customarily inserted by a compiler.

Examples
The following example demonstrates the use of the .es pseudo-op:
.. extern Pseudo-op

Purpose
Identifies a symbol defined in another source module.

Chapter 9. Pseudo-ops  481
Syntax
.extern  Name

Description
The .extern instruction identifies the Name value as a symbol defined in another source module, and Name becomes an external symbol. Any external symbols used in the current assembly that are not defined in the current assembly must be declared with an .extern statement. A locally defined symbol that appears in an .extern statement is equivalent to using that symbol in a .globl statement. A symbol not locally defined that appears in a .globl statement is equivalent to using that symbol in an .extern statement. An undefined symbol is flagged as an error unless the -u flag of the as command is used.

Parameters
Name Specifies an operand that is an external symbol and that can be a Qualname. (A Qualname parameter specifies the Name and StorageMappingClass values for the control section.)

Examples
The following example demonstrates the use of the .extern pseudo-op:
.extern proga[PR]
.toc
T.proga: .tc proga[TC],proga[PR]

Related Information
Pseudo-ops Overview
The .csect pseudo-op, .globl pseudo-op.

.file Pseudo-op

Purpose
Identifies a source file name.

Syntax
.file  StringConstant

Description
The .file pseudo-op provides symbol table information necessary for the use of the symbolic debugger and linkage editor. The .file pseudo-op also provides the intended target environment and source language type for the use of the link editor.

For cascade compilers, the .file pseudo-op has no other effect on assembly and is customarily inserted by the compiler.

It is recommended that the .file pseudo-op be placed at the beginning of the source code for assembly language programs. If the .file pseudo-op is omitted from the source code, the assembler processes the program as if the .file pseudo-op were the first statement. The assembler does this by creating an entry in the symbol table with the source program name as the file name. If the source is standard input, the file name will be noname. The assembler listing will not have this inserted entry.
Parameters

*StringConstant* Specifies the file name of the original source file.

Examples

1. To use a source file named *main.c*, enter:
   `.file "main.c"
2. To use a source file named *asml.s*, enter:
   `.file "asml.s"

Related Information

Pseudo-ops Overview

The `.function` pseudo-op.

.float Pseudo-op

Purpose
Stores a floating-point constant at the next fullword location.

Syntax

```
.float [FloatingConstant]
```

Description
The `.float` stores a floating-point constant at the next fullword location. Fullword alignment occurs if necessary.

Parameters

*FloatingConstant* Specifies a floating-point constant to be assembled.

Examples

The following example demonstrates the use of the `.float` pseudo-op:

```
.float 3.4
.float -77
,float 134E-12
```

Related Information

Pseudo-ops Overview

The `.double` pseudo-op.

.function Pseudo-op

Purpose
Identifies a function and provides additional information specific to the function.
Syntax

.function Name Expression1 Expression2 Expression3 | Expression4

Description

The .function pseudo-op identifies a function and provides symbol table information necessary for the use of the symbolic debugger.

The .function pseudo-op has no other effect on assembly and is customarily inserted by a compiler.

Parameters

Name Represents the function Name and should be defined as a symbol or control section (csect) Qualname in the current assembly. (A Qualname specifies a Name and StorageMappingClass for the control section.)

Expression1 Represents the top of the function.

Expression2 Represents the storage mapping class of the function.

Expression3 Represents the type of the function.

The third and fourth parameters to the .function pseudo-op serve only as place holders. These parameters are retained for downward compatibility with previous systems (RT, System V).

Expression4 Represents the size of the function (in bytes). This parameter must be an absolute expression. This parameter is optional.

Note: If the Expression4 parameter is omitted, the function size is set to the size of the csect to which the function belongs. A csect size is equal to the function size only if the csect contains one function and the beginning and end of the csect are the same as the beginning and end of the function.

Examples

The following example illustrates the use of the .function pseudo-op:

.globl .hello[pr]
csect .hello[pr]
.function .hello[pr],L.1B,16,044,0x86
L.1B:

Related Information

Pseudo-ops Overview

The .bf pseudo-op, .ef pseudo-op, .file pseudo-op.

.globl Pseudo-op

Purpose

Makes a symbol globally visible to the linker.

Syntax

.globl Name
Description
The `.globl` pseudo-op makes the symbol `Name` globally visible to the linker and available to any file that is linked to the file in which the `.globl` pseudo-op occurs.

- If the `.globl` pseudo-op is not used for a symbol, then that symbol is, unless otherwise effected, only visible within the current assembly and not to other modules that may later be linked to the current assembly. Alternately, the `.extern` or `.weak` pseudo-op can be used to effect visibility.
- If `Name` is defined in the current assembly, its type and value arise from that definition, not the `.globl` definition.
- The binder maps all common segments with the same name into the same memory. If the name is declared `.globl` and defined in one of the segments, this has the same effect as declaring the common symbols to be `.globl` in all segments. In this way, common memory can be initialized.

Parameters
`Name` Represents any label or symbol that is defined locally and requires external visibility. This parameter can be a `QualifiedName`. (A `QualifiedName` specifies a `Name` and `StorageMappingClass` for the control section.)

Examples
The following example illustrates the use of the `.globl` pseudo-op:
```assembly
.globl main
main:
  .csect data[rw]
  .globl data[rw]
```

Related Information
Pseudo-ops Overview

The `.comm` pseudo-op, `.extern` pseudo-op, and the `.weak` pseudo-op.

.hash Pseudo-op

Purpose
Associates a hash value with an external symbol.

Syntax
```
.hash Name StringConstant
```

Description
The hash string value contains type-checking information. It is used by the link-editor and program loader to detect variable mismatches and argument interface errors prior to the execution of a program.

Hash string values are usually generated by compilers of strongly typed languages. The hash value for a symbol can only be set once in an assembly. See [Type-Check Section](#) in the XCOFF Object (a.out) File Format for more information on type encoding and checking.

Parameters
`Name` Represents a symbol. Because this should be an external symbol, `Name` should appear in an `.extern` or `.global` statement.
StringConstant Represents a type-checking hash string value. This parameter consists of characters that represent a hexadecimal hash code and must be in the set [0-9A-F] or [0-9a-f].

A hash string comprises the following three fields:

- **Language Identifier** is a 2-byte field representing each language. The first byte is 0x00. The second byte contains predefined language codes that are the same as those listed in the `.source` pseudo-op.
- **General Hash** is a 4-byte field representing the most general form by which a data symbol or function can be described. It is the greatest common denominator among languages supported by AIX. A universal hash can be used for this field.
- **Language Hash** is a 4-byte field containing a more detailed, language-specified representation of data symbol or function.

Note: A hash string must have a length of 10 bytes. Otherwise, a warning message is reported when the `-w` flag is used. Since each character is represented by two ASCII codes, the 10-byte hash character string is represented by a string of 20 hexadecimal digits.

Examples
The following example illustrates the use of the `.hash` pseudo-op:

```
.extern b[pr]
.extern a[pr]
.extern e[pr]
.hash b[pr], "000A9375C1F51C2DCF0"
.hash a[pr], "ff0a2cc12365de30"  # warning may report
.hash e[pr], "000020202051C2DCF0"
```

Related Information
Pseudo-ops Overview

Type-Check Section in XCOFF Object (a.out) File Format.

The `.extern` pseudo-op, `.globl` pseudo-op.

.lcomm Pseudo-op

Purpose
Defines a local uninitialized block of storage.

Syntax

```
.lcomm Name1, Expression, Name2
```

Description
The `.lcomm` pseudo-op defines a local uninitialized block of storage called a local common (LC) section. At run time, this storage block will be reserved when the LC section is allocated at the end of the `.data` section. This storage block is for uninitialized data.

Use the `.lcomm` pseudo-op with local uninitialized data, which is data that will probably not be accessed outside the local assembly.
The symbol *Name1* is a label at the top of the local uninitialized block of storage. The location counter for this LC section is incremented by the *Expression* parameter. A specific LC section can be specified by the *Name2* parameter. Otherwise an unnamed section is used.

**Parameters**

*Name1* Represents a relocatable symbol. The symbol *Name1* is a label at the top of the local uninitialized block of storage. *Name1* does not appear in the symbol table unless it is the operand of a `.globl` statement.

*Expression* Represents an absolute expression that is defined in the first pass of the assembler. The *Expression* parameter also increments the location counter for the LC section.

*Name2* Represents a control section (csect) name that has storage mapping class BS and storage type CM. The *Name2* parameter allows the programmer to specify the BS csect for the allocated storage. If a specific LC section is not specified by the *Name2* parameter, an unnamed section is used.

**Examples**

1. To set up 5KB of storage and refer to it as buffer:

   ```
   .lcomm buffer,5120
   # Can refer to this 5K
   # of storage as "buffer".
   ```

2. To set up a label with the name proga:

   ```
   .lcomm b3,4,proga
   # b3 will be a label in a csect of class BS
   # and type CM with name "proga".
   ```

**Related Information**

[Pseudo-ops Overview](#)

The `.comm` pseudo-op.

---

**.lglobl Pseudo-op**

**Purpose**

Provides a means to keep the information of a static name in the symbol table.

**Syntax**

```
.lglobl  Name
```

**Description**

A static label or static function name defined within a control section (csect) must be kept in the symbol table so that the static label or static function name can be referenced. This symbol has a class of "hidden external" and differs from a global symbol. The `.lglobl` pseudo-op gives the symbol specified by the *Name* parameter have a symbol type of LD and a class of C_HIDEXT.

**Note:** The `.lglobl` pseudo-op does not have to apply to any csect name. The assembler automatically generates the symbol table entry for any csect name with a class of C_HIDEXT unless there is an explicit `.globl` pseudo-op applied to the csect name. If an explicit `.globl` pseudo-op applies to the csect name, the symbol table entry class for the csect is C_EXT.
Parameters

Name  Specifies a static label or static function name that needs to be kept in the symbol table.

Examples
The following example demonstrates the use of the .lglob pseudo-op:

```
.toc
.file  "test.s"
.lglobl  .foo
.csect  foo[DS]
foo:
.long   .foo,TOC[tc0],0
.csect  [PR]
.foo:
    .stabx  "foo:F-1",.foo,142,0
    .function  .foo,.foo,16,044,L..end_foo-.foo
```

Related Information

Pseudo-ops Overview

The .function pseudo-op, .globl pseudo-op.

=line Pseudo-op

Purpose
Identifies a line number and provides additional information specific to the line number.

Syntax

```
.line  [Number]
```

Description
The .line pseudo-op identifies a line number and is used with the .bi pseudo-op to provide a symbol table and other information necessary for use of the symbolic debugger.

This pseudo-op is customarily inserted by a compiler and has no other effect on assembly.

Parameters

Number  Represents a line number of the original source file.

Examples
The following example illustrates the use of the .line pseudo-op:

```
.globl  .hello[pr]
.csect  .hello[pr]
.align  1
.function  .hello[pr],L.1B,16,044
```
Related Information

Pseudo-ops Overview

The .bi pseudo-op, .bf pseudo-op, .function pseudo-op.

**.long Pseudo-op**

**Purpose**
Assembles expressions into consecutive fullwords.

**Syntax**

```
.long [Expression, Expression,...]
```

**Description**

The .long pseudo-op assembles expressions into consecutive fullwords. Fullword alignment occurs as necessary.

**Parameters**

Expression Represents any expression to be assembled into fullwords.

**Examples**

The following example illustrates the use of the .long pseudo-op:

```
.long 24,3,fooble-333,0
```

**Related Information**

Pseudo-ops Overview

The .byte pseudo-op, .short pseudo-op, .vbyte pseudo-op.

**.llong Pseudo-op**

**Purpose**
Assembles expressions into consecutive double-words.

**Syntax**

```
.llong [Expression, Expression,...]
```

**Description**

The .llong pseudo-op assembles expressions into consecutive double-words. In 32-bit mode, alignment occurs on fullword boundaries as necessary. In 64-bit mode, alignment occurs on double-word boundaries as necessary.
Parameters

Expression

Represents any expression to be assembled into fullwords/double-words.

Examples

The following example illustrates the use of the .llong pseudo-op:

```
.extern fooble
.llong 24,3,fooble-333,0
```

which fills 4 double-words, or 32 bytes, of storage.

Related Information

Pseudo-ops Overview

The .byte pseudo-op, .short pseudo-op, .vbyte pseudo-op, .long pseudo-op.

.machine Pseudo-op

Purpose

Defines the intended target environment.

Syntax

```
.machine StringConstant
```

Description

The .machine pseudo-op selects the correct instruction mnemonics set for the target machine. It provides symbol table information necessary for the use of the linkage editor. The .machine pseudo-op overrides the setting of the as command's -m flag, which can also be used to specify the instruction mnemonics set for the target machine.

The .machine pseudo-op can occur in the source program more than once. The value specified by a .machine pseudo-op overrides any value specified by an earlier .machine pseudo-op. It is not necessary to place the first .machine pseudo-op at the beginning of a source program. If no .machine pseudo-op occurs at the beginning of a source program and the -m flag is not used with the as command, the default assembly mode is used. The default assembly mode is overridden by the first .machine pseudo-op.

If a .machine pseudo-op specifies a value that is not valid, an error is reported. As a result, the last valid value specified by the default mode value, the -m flag, or a previous .machine pseudo-op is used for the remainder of the instruction validation in the assembler pass one.
Parameters

[StringConstant] Specifies the assembly mode. This parameter is not case-sensitive, and can be any of the values which can be specified with the -m flag on the command line. Possible values, enclosed in quotation marks, are:

**Null string ("") or nothing**
Specifies the default assembly mode. A source program can contain only instructions that are common to both POWER family and PowerPC. Any other instruction causes an error.

**push**
Saves the current assembly mode in the assembly mode pushdown stack.

**pop**
Removes a previously saved value from the assembly mode pushdown stack and restore the assembly mode to this saved value. **Note:** The intended use of **push** and **pop** is inside of include files which alter the current assembly mode. **.machine "push"** should be used in the included file, before it changes the current assembly mode with another **.machine**. Similarly, **.machine "pop"** should be used at the end of the included file, to restore the input assembly mode.

Attempting to hold more than 100 values in the assembly mode pushdown stack will result in an assembly error. The pseudo-ops **.machine "push"** and **.machine "pop"** are used in pairs.

**ppc**
Specifies the PowerPC common architecture, 32-bit mode. A source program can contain only PowerPC common architecture, 32-bit instructions. Any other instruction causes an error.

**ppc64**
Specifies the PowerPC 64-bit mode. A source program can contain only PowerPC common architecture, 32-bit instructions. Any other instruction causes an error.

**com**
Specifies the POWER family and PowerPC architecture intersection mode. A source program can contain only instructions that are common to both POWER family and PowerPC. Any other instruction causes an error.

**pwr**
Specifies the POWER family architecture, POWER family implementation mode. A source program can contain only instructions for the POWER family implementation of the POWER family architecture. Any other instruction causes an error.

**pwr2**
POWER family architecture, POWER2 implementation. A source program can contain only instructions for the POWER2 implementation of the POWER family architecture. Any other instruction causes an error. (**pwr2** is the preferred value, but the alternate value **pwrx** can also be used.)

**pwr5**
For AIX 5.3 and later, POWER family architecture, POWER5 implementation. A source program can contain only instructions for the POWER5 implementation of the POWER family architecture. Any other instruction causes an error.
any  Any nonspecific POWER family/PowerPC architecture or implementation mode. This includes mixtures of instructions from any of the valid architectures or implementations.

601  Specifies the PowerPC architecture, PowerPC 601 RISC Microprocessor mode. A source program can contain only instructions for the PowerPC architecture, PowerPC 601 RISC Microprocessor. Any other instruction causes an error.

Attention:  It is recommended that the 601 assembly mode not be used for applications that are intended to be portable to future PowerPC systems. The com or ppc assembly mode should be used for such applications.

The PowerPC 601 RISC Microprocessor implements the PowerPC architecture, plus some POWER family instructions which are not included in the PowerPC architecture. This allows existing POWER family applications to run with acceptable performance on PowerPC systems. Future PowerPC systems will not have this feature. The 601 assembly mode may result in applications that will not run on existing POWER family systems and that may not have acceptable performance on future PowerPC systems, because the 601 assembly mode permits the use of all the instructions provided by the PowerPC 601 RISC Microprocessor.

603  Specifies the PowerPC architecture, PowerPC 603 RISC Microprocessor mode. A source program can contain only instructions for the PowerPC architecture, PowerPC 603 RISC Microprocessor. Any other instruction causes an error.

604  Specifies the PowerPC architecture, PowerPC 604 RISC Microprocessor mode. A source program can contain only instructions for the PowerPC architecture, PowerPC 604 RISC Microprocessor. Any other instruction causes an error.

A35  Specifies the A35 mode. A source program can contain only instructions for the A35. Any other instruction causes an error.

Note:  See "as Command Flags" on page 53 for more information on assembly mode values.

Examples
1. To set the target environment to POWER family architecture, POWER family implementation:
    \[\text{.machine "pwr"}\]

2. To set the target environment to any non-specific POWER family/PowerPC architecture or implementation mode:
    \[\text{.machine "any"}\]

3. To explicitly select the default assembly mode:
    \[\text{.machine ""}\]

4. The following example of assembler output for a fragment of code shows the usage of .machine "push" and .machine "pop":

<table>
<thead>
<tr>
<th>File#</th>
<th>Line#</th>
<th>Mode</th>
<th>Name</th>
<th>Loc</th>
<th>Ctr</th>
<th>Object Code</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>PWR2</td>
<td>longna</td>
<td>00000000</td>
<td>00000000a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>PWR2</td>
<td>longna</td>
<td>00000004</td>
<td>329e000a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>PWR2</td>
<td>longna</td>
<td>00000008</td>
<td>81540014</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>9</td>
<td>PPC</td>
<td>a2</td>
<td>00000000</td>
<td>7d4c42e6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>PWR2</td>
<td>a2</td>
<td>00000004</td>
<td>329e000a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

492  Assembler Language Reference
**Related Information**

“Host Machine Independence and Target Environment Indicator Flag” on page 4.

“Pseudo-ops Overview” on page 463.

“Assembling with the as Command” on page 53.

---

**.org Pseudo-op**

**Purpose**
Sets the value of the current location counter.

**Syntax**

```
.org Expression
```

**Description**

The `.org` pseudo-op sets the value of the current location counter to `Expression`. This pseudo-op can also decrement a location counter. The assembler is control section (csect) oriented; therefore, absolute expressions or expressions that cause the location counter to go outside of the current csect are not allowed.

**Parameters**

`Expression` Represents the value of the current location counter.

**Examples**

The following example illustrates the use of the `.org` pseudo-op:

```assembly
# Assume assembler location counter is 0x114.
.org $+100
# Skip 100 decimal byte (0x64 bytes).
.
.
# Assembler location counter is now 0x178.
```

---

**Related Information**

Pseudo-ops Overview

The `.space` pseudo-op.

---

**.quad Pseudo-op**

**Purpose**
Stores a quad floating-point constant at the next fullword location. Alignment requirements for floating-point data are consistent between 32-bit and 64-bit modes.

**Syntax**

```
.quad FloatingConstant
```
Examples
The following example demonstrates the use of the .quad pseudo-op:

```
.quad 3.4
.quad -77
.quad 134e12
.quad 5e300
.quad 0.45
```

The above declarations would reserve 16 bytes of storage each.

Related Information
Pseudo-ops Overview

The .float pseudo-op, .double pseudo-op.

---

.ref Pseudo-op

Purpose
Creates a R_REF type entry in the relocation table for one or more symbols.

Syntax
```
.ref [Name[,Name...]]
```

Description
The .ref pseudo-op supports the creation of multiple RLD entries in the same location. This pseudo-op is used in the output of some compilers to ensure the linkage editor does not discard routines that are used but not referenced explicitly in the text or data sections.

For example, in C++, constructors and destructors are used to construct and destroy class objects. Constructors and destructors are sometimes called only from the run-time environment without any explicit reference in the text section.

The following rules apply to the placement of a .ref pseudo-op in the source program:
- The .ref pseudo-op cannot be included in a dsect or csect with a storage mapping class of BS or UC.
- The .ref pseudo-op cannot be included in common sections or local common sections.

The following rules apply to the operands of the .ref pseudo-op (the Name parameter):
- The symbol must be defined in the current source module.
- External symbols can be used if they are defined by .extern or .globl.
- Within the current source module, the symbol can be a csect name (meaning a Qualname) or a label defined in the csect.
- The following symbols cannot be used for the .ref operand:
  - pseudo-op .dsect names
  - labels defined within a dsect
  - a csect name with a storage mapping class of BS or UC
  - labels defined within a csect with a storage mapping class of BS or UC
  - a pseudo-op .set Name operand which represents a non-relocatable expression type
Parameters

*Name*  Specifies a symbol for which a R_REF type entry in the relocation table should be created.

Examples

The following example demonstrates the use of the `.ref` pseudo-op:

```
C1: .csect a1[pr]
   .csect a2[pr]
   .set r10,10
   .extern C4
C2: .long 10
C3: .long 20
   .ref C1,C2,C3
   .ref C4
```

Related Information

[Combination Handling of Expressions](#) (This discusses another way to generate a R_REF type entry in the relocation table.)

.rename Pseudo-op

Purpose

Creates a synonym or alias for an illegal or undesirable name.

Syntax

```
.rename Name, StringConstant
```

Description

The restrictions on the characters that can be used for symbols within an assembler source file are defined in ["Constructing Symbols" on page 31](#). The symbol cannot contain any blanks or special characters, and cannot begin with a digit.

For any external symbol that must contain special characters, the `.rename` pseudo-op provides a way to do so.

The `.rename` pseudo-op changes the *Name* parameter to the *StringConstant* value for all external references at the end of assembly. Internal references to the local assembly are made to *Name*. The externally visible *Name* is *StringConstant*. The `.rename` pseudo-op is useful in referencing symbol names that are otherwise illegal in the assembler syntax.

Parameters

*Name*  Represents a symbol. To be externally visible, the *Name* parameter must appear in an `.extern` or `.globl` statement.

*StringConstant*  Represents the value to which the *Name* parameter is changed at end of assembly.
Examples

The following example illustrates the use of the .rename pseudo-op:

```
csect mst_sect[RW]
globl mst_sect[RW]
OK_chars:
globl OK_chars
long OK_chars
.rename OK_chars,$SPECIAL$_char
.rename mst_sect[RW],"MST_sect_renamed"
```

Related Information

"Pseudo-ops Overview" on page 463.
"Constructing Symbols" on page 31.
".extern Pseudo-op" on page 481, ".globl Pseudo-op" on page 484.

.set Pseudo-op

Purpose

Sets a symbol equal to an expression in both type and value.

Syntax

```
.set  Name, Expression
```

Description

The .set pseudo-op sets the Name symbol equal to the Expression value in type and in value. Using the .set pseudo-op may help to avoid errors with a frequently used expression. Equate the expression to a symbol, then refer to the symbol rather than the expression. To change the value of the expression, only change it within the .set statement. However, reassembling the program is necessary since .set assignments occur only at assembly time.

The Expression parameter is evaluated when the assembler encounters the .set pseudo-op. This evaluation is done using the rules in Combination Handling of Expressions; and the type and value of the evaluation result are stored internally. If evaluating the Expression results in an invalid type, all instructions which use the symbol Name will have an error.

The stored type and value for symbol Name, not the original expression definition, are used when Name is used in other instructions.

Parameters

- **Name**: Represents a symbol that may be used before its definition in a .set statement; forward references are allowed within a module.

- **Expression**: Defines the type and the value of the symbol Name. The symbols referenced in the expression must be defined; forward references are not allowed. The symbols cannot be undefined external expressions. The symbols do not have to be within the control section where the .set pseudo-op appears. The Expression parameter can also refer to a register number, but not to the contents of the register at run time.
Examples
1. The following example illustrates the use of the .set pseudo-op:
   ```
   .set ap,14  # Assembler assigns value 14
   # to the symbol ap -- ap
   # is absolute.
   .
   l1l ap,2
   # Assembler substitutes value 14
   # for the symbol.
   # Note that ap is a register
   # number in context
   # as l1l's operand.
   ```
2. The following example will result in an assembly error because of an invalid type:
   ```
   .csect a1[PR]
   L1:  l  20,30(10)
   .csect a2[rw]
   .long 0x20
   L2: .long 0x30
   .set r1, L2 - L1  # r1 has type of E_REXT
   # r1 has value of 8
   .long r1 + 10
   .long L2 - r1  # Error will be reported.
   # L2 is E_REL
   # r1 is E_REXT
   # E_REL - E_REXT ==> Invalid type
   ```

Related Information
Pseudo-ops Overview
Expressions

.short Pseudo-op

Purpose
Assembles expressions into consecutive halfwords.

Syntax
```
.short [Expression,Expression,...]
```

Description
The .short pseudo-op assembles `Expressions` into consecutive halfwords. Halfword alignment occurs as necessary.

Parameters
- `Expression`: Represents expressions that the instruction assembles into halfwords. The `Expression` parameter cannot refer to the contents of any register. If the `Expression` value is longer than a halfword, it is truncated on the left.
Examples
The following example illustrates the use of the .short pseudo-op:
.short 1,0x4444,fooble-333,0

Related Information
Pseudo-ops Overview

The .byte pseudo-op, .long pseudo-op, .vbyte pseudo-op.

.source Pseudo-op

Purpose
Identifies the source language type.

Syntax
[source]  [StringConstant]

Description
The .source pseudo-op identifies the source language type and provides symbol table information necessary for the linkage editor. For cascade compilers, the symbol table information is passed from the compiler to the assembler to indicate the high-level source language type. The default source language type is “Assembler.”

Parameters
[StringConstant] Specifies a valid program language name. This parameter is not case-sensitive. If the specified value is not valid, the language ID will be reset to “Assembler.” The following values are defined:
0x00 C
0x01 FORTRAN
0x02 Pascal
0x03 Ada
0x04 PL/1
0x05 BASIC
0x06 LISP
0x07 COBOL
0x08 Modula2
0x09 C++
0x0a RPG
0x0b PL8, PLIX
0x0c Assembler
Examples
To set the source language type to C++:
    .source "C++"

Related Information
Pseudo-ops Overview
Source Language Type

.space Pseudo-op

Purpose
Skips a specified number of bytes in the output file and fills them with binary zeros.

Syntax
    .space  Number

Description
The .space skips a number of bytes, specified by Number, in the output file and fills them with binary zeros. The .space pseudo-op may be used to reserve a chunk of storage in a control section (csect).

Parameters
Number Represents an absolute expression that specifies the number of bytes to skip.

Examples
The following example illustrates the use of the .space pseudo-op:
    .csect data[rw]
    .space 444
    
    foo:  # foo currently located at offset 0x1BC within
         # csect data[rw].

Related Information
Pseudo-ops Overview

.stabx Pseudo-op

Purpose
Provides additional information required by the debugger.

Syntax
    .stabx  StringConstant Expression1 Expression2 Expression3
Description
The .stabx pseudo-op provides additional information required by the debugger. The assembler places the StringConstant argument, which provides required stabstring information for the debugger, in the .debug section.

The .stabx pseudo-op is customarily inserted by a compiler.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringConstant</td>
<td>Provides required Stabstring information to the debugger.</td>
</tr>
<tr>
<td>Expression1</td>
<td>Represents the symbol value of the character string. This value is storage mapping class dependent. For example, if the storage mapping class is C_LSYM, the value is the offset related to the stack frame. If the storage mapping class is C_FUN, the value is the offset within the containing control section (csect).</td>
</tr>
<tr>
<td>Expression2</td>
<td>Represents the storage class of the character string.</td>
</tr>
<tr>
<td>Expression3</td>
<td>Represents the symbol type of the character string.</td>
</tr>
</tbody>
</table>

Examples
The following example illustrates the use of the .stabx pseudo-op:

```
.stabx "INTEGER:t2=-1",0,140,4
```

Related Information

- Pseudo-ops Overview
- Debug Section in the XCOFF Object (a.out) File Format.

The .function pseudo-op.

---

.string Pseudo-op

Purpose
Assembles character values into consecutive bytes and terminates the string with a null character.

Syntax

```
.string StringConstant
```

Description
The .string pseudo-op assembles the character values represented by StringConstant into consecutive bytes and terminates the string with a null character.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringConstant</td>
<td>Represents a string of character values assembled into consecutive bytes.</td>
</tr>
</tbody>
</table>

Examples
The following example illustrates the use of the .string pseudo-op:

```
mine: .string "Hello, world!"
# This produces
# 0x48656c6c6f2c20776f726c642100.
```
.ttag Pseudo-op

Purpose
Defines a debug traceback tag, preceded by a word of zeros, that can perform tracebacks for debugging programs.

Syntax
.ttag [Expression1, Expression2, Expression3, Expression4, Expression5, Expression6, Expression7, Expression8, Expression9, Expression10, Expression11, Expression12, Expression13, Expression14, Expression15, Expression16]

Description
The .ttag pseudo-op defines a traceback tag by assembling Expressions into consecutive bytes, words, and halfwords, depending on field requirements. An instruction can contain either 8 expressions (Expression1 through Expression8) or 16 expressions (Expression1 through Expression16). Anything else is a syntax error. A compiler customarily inserts the traceback information into a program at the end of the machine instructions, adding a string of zeros to signal the start of the information.

Parameters

<table>
<thead>
<tr>
<th>Expression1</th>
<th>version</th>
<th>/*Traceback format version */</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression2</th>
<th>lang</th>
<th>/*Language values */</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|                      | TB_C        | 0                             |
|                      | TB_FORTRAN  | 1                             |
|                      | TB_PASCAL   | 2                             |
|                      | TB_ADA      | 3                             |
|                      | TB_PL1      | 4                             |
|                      | TB_BASIC    | 5                             |
|                      | TB_LISP     | 6                             |
|                      | TB_COBOL    | 7                             |
|                      | TB_MODULA2  | 8                             |
|                      | TB_CPLUSPLUS| 9                             |
|                      | TB_RPG      | 10                            |
|                      | TB_PL8      | 11                            |
|                      | TB_ASM      | 12                            |

<table>
<thead>
<tr>
<th>Expression3</th>
<th>/*Traceback control bits */</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td></td>
</tr>
</tbody>
</table>

| globallink    | Bit 7. Set if routine is global linkage. |
| is_eprol      | Bit 6. Set if out-of-town epilog/prologue. |
| has_tboff     | Bit 5. Set if offset from start of proc stored. |
| int_proc      | Bit 4. Set if routine is internal. |
| has_ctl       | Bit 3. Set if routine involves controlled storage. |
| toless        | Bit 2. Set if routine contains no TOC. |
| fp_present    | Bit 1. Set if routine performs FP operations. |
| log_abort     | Bit 0. Set if routine involves controlled storage. |

<table>
<thead>
<tr>
<th>Expression4</th>
<th>/*Traceback control bits (continued) */</th>
</tr>
</thead>
</table>
Expressions

Examples

The following example illustrates the use of the .tbtag pseudo-op:

```
.tbtag 1,0,0xff,0,0,16,0,0
```

Related Information

"Pseudo-ops Overview" on page 463.

"Traceback Tags" on page 78.
.tc Pseudo-op

Purpose
Assembles expressions into a Table of Contents (TOC) entry.

Syntax
.tc \[Name\[TC\], Expression, Expression,...\]

Note: The boldface brackets containing TC are part of the syntax and do not specify optional parameters.

Description
The .tc pseudo-op assembles Expressions into a TOC entry, which contains the address of a routine, the address of a function descriptor, or the address of an external variable. A .tc statement can only appear inside the scope of a .toc pseudo-op. A TOC entry can be relocated as a body. TOC entry statements can have local labels, which will be relative to the beginning of the entire TOC as declared by the first .toc statement. Addresses contained in the TOC entry can be accessed using these local labels and the TOC Register GPR 2.

TOC entries that contain only one address are subject to being combined by the binder. This can occur if the TOC entries have the same name and reference the same control section (csect) (symbol). Be careful when coding TOC entries that reference nonzero offsets within a csect. To prevent unintended combining of TOC entries, unique names should be assigned to TOC entries that reference different offsets within a csect.

Parameters
Name Specifies name of the TOC entry created. The StorageMappingClass is TC for TOC entries. Name[TC] can be used to refer to the TOC entry where appropriate.
Expression Specifies symbol or expression which goes into TOC entry.

Examples
The following example illustrates the use of the .tc pseudo-op:

.toc
  \# Create three TOC entries, the first
  \# with the name proga, the second
  \# with the name progB, and the last
  \# unnamed.
  T.proga: .tc proga[TC], progr[RW], dataA
  T.progb: .tc progB[TC], progb[PR], progB[PR]
  T.progax: .tc proga[TC], dataB
  .tc [TC], dataB
  .csect proga[PR]
  \# A .csect should precede any statements following a
  \# .toc/.tc section which do not belong in the TOC.
  l 5,T.proga(2) \# The address of progr[RW]
  \# is loaded into GPR 5.
  l 5,T.progax(2) \# The address of progr[RW]
  \# is loaded into GPR 5.
  l 5,T.progb+4(2) \# The address of progB[PR]
  \# is loaded into GPR 5.
.toc Pseudo-op

Purpose
Defines the table of contents of a module.

Syntax
.toct

Description
The .toc pseudo-op defines the table of contents (TOC) anchor of a module. Entries in the TOC section can be declared with .tc pseudo-op within the scope of the .toc pseudo-op. The .toc pseudo-op has scope similar to that of a .csect pseudo-op. The TOC can be continued throughout the assembly wherever a .toc appears.

Examples
The following example illustrates the use of the .toc pseudo-op:

```
.toc
# Create two TOC entries. The first entry, named proga,
# is of type TC and contains the address of proga[RW] and dataA.

# The second entry, named progb, is of type TC and contains
# the address of progb[PR] and progc[PR].
T.proga: .tc proga[TC],proga[RW],dataA
T.progb: .tc progb[TC],progb[PR],progc[PR]

.csect proga[RW]
```

Related Information
The .tc pseudo-op, .tocof pseudo-op.

.tocof Pseudo-op

Purpose
Allows for the definition of a local symbol as the table of contents of an external symbol so that the local symbol can be used in expressions.

Syntax
.tocof Name1 Name2
Description

The .tocf pseudo-op makes the Name2 value globally visible to the linker and marks the Name1 symbol as the table of contents (TOC) of another module that contains the symbol Name2. As a result, a local symbol can be defined as the TOC of an external symbol so that the local symbol can be used in expressions or to refer to the TOC of another module, usually in a .tc statement. This pseudo-op generates a Relocation Dictionary entry (RLD) that causes this data to be initialized to the address of the TOC external symbols. The .tocf pseudo-op can be used for intermodule calls that require the caller to first load up the address of the called module’s TOC before transferring control.

Parameters

Name1 Specifies a local symbol that acts as the TOC of a module that contains the Name2 value. The Name1 symbol should appear in .tc statements.

Name2 Specifies an external symbol that exists within a module that contains a TOC.

Examples

The following example illustrates the use of the .tocf pseudo-op:

tocbeg: .toc
abp: .tc [tc],pb,tpb
# This is an unnamed TOC entry
# that contains two addresses:
# the address of pb and
# the address of the TOC
# containing pb.
.tocf tpb,pb
.set always,0x14
.csect [PR]
.using tocbeg,rtoc
l 14,apb
# Load R14 with the address
# of pb.
1 rtoc,apb+4
# Load the TOC register with the
# address pb's TOC.
mtspr lr,14
# Move to Link Register.
bcr always,0
# Branch Conditional Register branch
# address is contained in the Link
# register.

Related Information

“Pseudo-ops Overview” on page 463.

“Understanding and Programming the TOC” on page 82.

“.tc Pseudo-op” on page 503,.toc Pseudo-op” on page 504.

.using Pseudo-op

Purpose

Allows the user to specify a base address and assign a base register number.

Syntax

.using Expression Register
Description

The .using pseudo-op specifies an expression as a base address, and assigns a base register, assuming that the Register parameter contains the program address of Expression at run time. Symbol names do not have to be previously defined.

**Note:** The .using pseudo-op does not load the base register; the programmer should ensure that the base address is loaded into the base register before using the implicit address reference.

The .using pseudo-op only affects instructions with an implicit-based address. It can be issued on the control section (csect) name and all labels in the csects. It can also be used on the dsect name and all the labels in the dsects. Other types of external symbols are not allowed (.extern).

Using Range

The range of a .using pseudo-op (using range) is -32768 or 32767 bytes, beginning at the base address specified in the .using pseudo-op. The assembler converts each implicit address reference (or expression), which lies within the using range, to an explicit-based address form. Errors are reported for references outside the using range.

Two using ranges overlap when the base address of one .using pseudo-op lies within the ranges of another .using pseudo-op. When using range overlap happens, the assembler converts the implicit address reference by choosing the smallest signed offset from the base address as the displacement. The corresponding base register is used in the explicit address form. This applies only to implicit addresses that appear after the second .using pseudo-op.

In the next example, the using range of base2 and data[PR] overlap. The second l instruction is after the second .using pseudo-op. Because the offset from data[PR] to d12 is greater than the offset from base2 to d12, base2 is still chosen.

```
.csect data[PR]
  .long 0x1
dl:
  .long 0x2
base2:
  .long 0x3
  .long 0x4
  .long 0x4
  .long 0x5
d12:
  .long 0x6
  l 12, data_block.T(2)  # Load addr. of data[PR] into r12
cal 14, base2(12)      # Load addr. of base2 into r14
  .using base2, 14
  l 4, d12             # Convert to 1 4, 0xc(14)
  .using data[PR], 12
  l 4, d12            # Converts to 1 4, 0xc(14)
               # because base2 is still chosen
.toe
```

data_block.T: tc data_block[tc], data[PR]

There is an internal using table that is used by the assembler to track the .using pseudo-op. Each entry of the using table points to the csect that contains the expression or label specified by the Expression parameter of the .using pseudo-op. The using table is only updated by the .using pseudo-ops. The location of the .using pseudo-ops in the source program influences the result of the conversion of an implicit-based address. The next two examples illustrate this conversion.

**Example 1:**

```
  .using label1,4
  .using label2,5
  .csect data[RW]
label1:
  .long label1
  .long label2
```

506  Assembler Language Reference
.long 8
label1_a: .long 16
   .long 20
label2: .long label2
   .long 28
   .long 32
label2_a: .long 36
   .long 40
.csect sub1[pr]
 1 6,label1_a  # base address label2 is
     # chosen, so convert to:
     # 1 6, -8(5)
 1 6,label2_a  # base address label2 is
     # chosen, so convert to:
     # 1 6, 0xc(5)

Example 2:
.csect data[Rw]
label1: .long label1
   .long label2
   .long 12
label1_a: .long 16
   .long 20
label2: .long label2
   .long 28
   .csect sub2[pr]
   .using label1,4
 1 6,label1_a  # base address label1 is
     # chosen, so convert to:
     # 1 6, 0xc(4)
   .using label2,5
 1 6,label1_a  # base address label2 is
     # chosen, so convert to:
     # 1 6, -8(5)

Two using ranges coincide when the same base address is specified in two different .using pseudo-ops, while the base register used is different. The assembler uses the lower numbered register as the base register when converting to explicit-based address form, because the using table is searched from the lowest numbered register to the highest numbered register. The next example shows this case:

.csect data[PR]
   .long 0x1
d1:   .long 0x2
base2; .long 0x3
   .long 0x4
   .long 0x5
d12:  .long 0x6
 1 12, data_block.T(2)  # Load addr. of data[PR] into r12
 1 14, data_block.T(2)  # Load addr. of data[PR] into r14
   .using data[PR], 12
 1 4, d12  # Convert to: 1 4, 0x14(12)
   .using data[PR], 14
 1 4, d12  # Convert to: 1 4, 0x14(12)
   .toc
data_block.T: .tc data_block[tc], data[PR]

Using Domain
The domain of a .using pseudo-op (the using domain) begins where the .using pseudo-op appears in a csect and continue to the end of the source module except when:

- A subsequent .drop pseudo-op specifies the same base register assigned by the preceding .using pseudo-op.
- A subsequent .using pseudo-op specifies the same base register assigned by the preceding .using pseudo-op.
These two exceptions provide a way to use a new base address. The next two examples illustrate these exceptions:

**Example 1:**
```
.csect data[PR]
.long 0x1
.dl: .long 0x2
base2: .long 0x3
.long 0x4
.long 0x5
dl2: .long 0x6
    1 12, data_block.T(2)  # Load addr. of data[PR] into r12
    1 14, base2(12)       # Load addr. of base2 into r14
    4, dl2                # Convert to: 1 4, 0xc(14)
    1 14, data_block.T(2) # Load addr. of data[PR] into r14
    1 4, dl2              # Convert to: 1 4, 0x14(14)
    .using base2, 14
    .drop dl2             # base address base2 is used
    1 4, dl2              # Convert to: 1 4, 0x14(14)
    .toc
```

data_block.T: .tc data_block[tc], data[PR]

**Example 2:**
```
.csect data[PR]
.long 0x1
.dl: .long 0x2
base2: .long 0x3
.long 0x4
.long 0x5
dl2: .long 0x6
    1 12, data_block.T(2)  # Load addr. of data[PR] into r12
    1 14, base2(12)       # Load addr. of base2 into r14
    4, dl2                # Convert to: 1 4, 0xc(14)
    1 4, dl2              # Convert to: 1 4, 0x14(14)
    .using data[PR], 12
    .drop dl2             # base address base2 is used
    1 4, dl2              # Convert to: 1 4, 0x14(14)
    .toc
```

data_block.T: .tc data_block[tc], data[PR]

**Note:** The assembler does not convert the implicit address references that are outside the Using Domain. So, if these implicit address references appear before any .using pseudo-op that defines a base address of the current csect, or after the .drop pseudo-ops drop all the base addresses of the current csect, an error is reported.

The next example shows the error conditions:
```
.csect data[PR]
.long 0x1
.dl: .long 0x2
base2: .long 0x3
.long 0x4
.long 0x5
dl2: .long 0x6
    4, dl2                # Error is reported here
    1 12, data_block.T(2)  # Load addr. of data[PR] into r12
    1 14, data_block.T(2) # Load addr. of data[PR] into r14
    1 4, dl2              # base address base2 is used
    1 14, data_block.T(2) # Load addr. of data[PR] into r14
    .using data[PR], 12
    1 4, dl2              # Error is reported here
    .using data[PR], 14
    .drop dl2             # base address base2 is used
    1 4, dl2
```
Parameters

Register

Represents the register number for expressions. It must be absolute and must evaluate to an integer from 0 to 31 inclusive.

Expression

Specifies a label or an expression involving a label that represents the displacement or relative offset into the program. The Expression parameter can be an external symbol if the symbol is a csect or Table of Contents (TOC) entry defined within the assembly.

Examples

The following example demonstrates the use of the .using pseudo-op:

```assembly
.csect data[rw]
.long 0x0, 0x0
d1: .long 0x25
# A read/write csect contains the label d1.
.csect text[pr]
.using data[rw], 12
1 4,d1
# This will actually load the contents of
# the effective address, calculated by
# adding the address d1 to the address in
# GPR 12, into GPR 4
```

Related Information

"Pseudo-ops Overview" on page 463.
"Implicit-Based Addressing" on page 50.

.vbyte Pseudo-op

Purpose

Assembles the value represented by an expression into consecutive bytes.

Syntax

```assembly
.vbyte Number Expression
```

Description

The .vbyte pseudo-op assembles the value represented by the Expression parameter into a specified number of consecutive bytes.
Parameters

Number Specifies a number of consecutive bytes. The Number value must range between 1 and 4.

Expression Specifies a value that is assembled into consecutive bytes. The Expression parameter cannot contain externally defined symbols. If the Expression value is longer than the specified number of bytes, it will be truncated on the left.

Examples
The following example illustrates the use of the .vbyte pseudo-op:

```assembly
.csect data[RW]
mine: .vbyte 3,0x37CCFF
# This pseudo-op also accepts character constants.
.vbyte 1,'c
# Load GPR 4 with address of .csect data[RW].
.csect text[PR]
 3,mine(4)
# GPR 3 now holds 0x37CCFF.
```

Related Information
Pseudo-ops Overview

The .byte pseudo-op.

.weak Pseudo-op

Purpose
Makes a symbol with weak binding globally visible to the linker.

Syntax
```
.weak Name
```

Description
The .weak pseudo-op makes the symbol Name globally visible to the linker and available to any file that is linked to the file in which either the .globl or .weak pseudo-op occurs. However, the symbol has weak binding semantics.

- If the .weak pseudo-op is not used for a symbol, then that symbol is, unless otherwise effected, only visible within the current assembly and not to other modules that may later be linked to the current assembly.
- If Name is defined in the current assembly, its type and value arise from that definition, not the .weak definition.
- Once .weak has been seen for a symbol, latter occurrances of .globl and .extern will not affect it for that file.
- The binder ignores duplicate definitions for symbols with the same name that are weak. If the name is declared .globl in one object file of module, and .weak in another, the global definition is used and the weak ones are ignored. If no global definition (such as the C_EXT storage class ) exists, the first weak definition is used, according to link order as described by the ld reference page.
Parameters

Name Represents any label or symbol that is defined locally and requires external visibility with weak storage class. This parameter can be a Qualname. (A Qualname specifies a Name and StorageMappingClass for the control section.)

Examples
The following example illustrates the use of the .weak pseudo-op:

```
.weak foo[RW]
csect data[RW]
```

Related Information

- [Pseudo-ops Overview](#)
- The .globl and .extern pseudo-ops.
- The ld command.

.xline Pseudo-op

Purpose
Represents a line number.

Syntax
```
.xline Number1, StringConstant [ , Number2 ]
```

Description
The .xline pseudo-op provides additional file and line number information to the assembler. The Number2 parameter can be used to generate .bi and .ei type entries for use by symbolic debuggers. This pseudo-op is customarily inserted by the M4 macro processor.

Parameters

- Number1 Represents the line number of the original source file.
- StringConstant Represents the file name of the original source file.
- Number2 Represents the C_BINCL and C_EINCL classes, which indicate the beginning and ending of an included file, respectively.

Examples
The following example illustrates the use of the .xline pseudo-op:

```
.xline 1,"hello.c",108
.xline 2,"hello.c"
```

Related Information

[Pseudo-ops Overview](#)
Appendix A. Messages

The messages in this appendix are error messages or warning messages. Each message contains three sections:

- Message number and message text
- Cause of the message
- Action to be taken

For some messages that are used for file headings, the Action section is omitted.

1252-001  
<name> is defined already.

**Cause** The user has previously used `name` in a definition-type statement and is trying to define it again, which is not allowed. There are three instances where this message is displayed:

- A label name has been defined previously in the source code.
- A `.set` pseudo-op name has been defined previously in the source code.
- A `.lcomm` or `.comm` pseudo-op name has been previously defined in the source code.

**Action** Correct the name-redefined error.

1252-002  
There is nesting overflow. Do not specify more than 100 `.function`, `.bb`, or `.bi` pseudo-ops without specifying the matching `.ef`, `.eb`, or `.ei` pseudo-ops.

**Cause** This syntax error message will only be displayed if debugger pseudo-ops are used. The `.function`, `.bb`, and `.bi` pseudo-ops generate pointers that are saved on a stack with a limiting size of 100 pointers. If more than 100 `.function` and `.bb` pseudo-ops have been encountered without encountering the matching `.ef` and `.eb` pseudo-ops, this syntax error message is displayed.

**Action** Rewrite the code to avoid this nesting.

**Note:** Debugger pseudo-ops are normally generated by compilers, rather than being inserted in the source code by the programmer.

1252-003  
The `.set` operand is not defined or is a forward reference.

**Cause** The `.set` pseudo-op has the following syntax:

```
.set name,expr
```

The `expr` parameter can be an integer, a predefined name (specified by a label, or by a `.lcomm` or `.comm` pseudo-op) or an algebraic combination of an integer and a name. This syntax error message appears when the `expr` parameter is not defined.

**Action** Verify that all elements of the `expr` parameter are defined before the `.set` statement.
1252-004  The .globl symbol is not valid. Check that the .globl name is a relocatable expression.

Cause  The .globl name must be a relocatable expression. This syntax error message is displayed when the Name parameter of the .globl pseudo-op is not a relocatable expression.

Relocation refers to an entity that represents a memory location whose address or location can and will be changed to reflect run-time locations. Entities and symbol names that are defined as relocatable or nonrelocatable are described in "Expressions" on page 39.

Relocatable expressions include label names, .lcomm, .comm names, and .csect names.

The following are the nonrelocatable items and nonrelocatable expressions:
- .dsect names
- labels contained within a .dsect
- labels contained within a csect with a storage class of BS or UC
- .set names
- absolute expression (constant or integer)
- tocrelative (.tc label or name)
- tocofrelative (.tocof label or name)
- unknown (undefined in Pass 2 of the assembler)

Action  Ensure that the Name parameter of the .globl pseudo-op is a relocatable expression. If not defined, the name is assumed to be external.

1252-005  The storage class is not valid. Specify a supported storage class for the csect name.

Cause  This syntax error message is displayed when the storage mapping class value used to specify the Qualname in the .csect pseudo-op is not one of the predefined values.

Action  See the .csect pseudo-op for the list of predefined storage mapping classes. Correct the program error and assemble and link the program again.

1252-006  The ERRTOK in the ICSECT ERRTOK is not known. Depending upon where you acquired this product, contact either your service representative or your approved supplier.

Cause  This is an internal error message.

Action  Contact your service representative or your approved supplier to report the problem.

1252-007  The alignment must be an absolute expression.

Cause  This syntax error message is caused by an incorrect operand (the optional alignment parameter) to the .csect pseudo-op. This alignment parameter must be either an absolute expression (an integer) or resolve algebraically into an absolute expression.

Action  Correct the alignment parameter, then assemble and link the program again.

1252-008  The .tocof name1 is not valid. Check that the name1 has not been defined previously.

Cause  The Name1 parameter of the .tocof pseudo-op has been defined elsewhere in the current module.

Action:  Ensure that the name1 symbol is defined only in the .tocof pseudo-op.

1252-009  A Begin or End block or .function pseudo-op is missing. Make sure that there is a matching .eb statement for each .bb statement and that there is a matching .ef statement for each .bf statement.

Cause  If there is not a matching .eb pseudo-op for each .bb pseudo-op or if there is not a matching .ef pseudo-op for each .bf pseudo-op, this error message is displayed.

Action  Verify that there is a matching .eb pseudo-op for every .bb pseudo-op, and verify that there is a matching .ef pseudo-op for every .bf pseudo-op.
1252-010 The .tcof Name2 is not valid. Make sure that name2 is an external symbol.

**Cause** The Name2 parameter for the .tcof pseudo-op has not been properly defined.

**Action** Ensure that the Name2 parameter is externally defined (it must appear in an extern or .globl pseudo-op) and ensure that it is not defined locally in this source module.

**Note:** If the Name2 parameter is defined locally and is externalized using a .extern pseudo-op, this message is also displayed.

1252-011 A .space parameter is undefined.

**Cause** The Number parameter to the .space pseudo-op must be a positive absolute expression. This message indicates that the Number parameter contains an undefined element (such as a label or name for a .lcomm, .comm, or .csect pseudo-op that will be defined later).

**Action** Verify that the Number parameter is an absolute expression, integer expression, or an algebraic expression that resolves into an absolute expression.

1252-012 The .space size must be an absolute expression.

**Cause** The Number parameter to the .space pseudo-op must be a positive absolute expression. This message indicates that the Number parameter contains a nonabsolute element (such as a label or name for a .lcomm, .comm, or .csect pseudo-op).

**Action** Verify that the Number parameter specifies an absolute expression, or an integer or algebraic expression that resolves into an absolute expression.

1252-013 The .space size must be a positive absolute expression.

**Cause** The Number parameter to the .space pseudo-op must be a positive absolute expression. This message indicates that the Number parameter resolves to a negative absolute expression.

**Action** Verify that the Number parameter is a positive absolute expression.

1252-014 The .rename Name symbol must be defined in the source code.

**Cause** The Name parameter to the .rename pseudo-op must be defined somewhere in the source code. This message indicates that the Name parameter has not been defined.

**Action** Verify that the Name parameter is defined somewhere in the source code.

1252-015 A pseudo-op parameter is not defined.

**Cause** This is a syntax error message displayed for the .line, .xline, .bf, .ef, .bb and .eb pseudo-ops. These expressions have an expression operand that must resolve.

**Action** Change the source code so that the expression resolves or is defined.

1252-016 The specified opcode or pseudo-op is not valid. Use supported instructions or pseudo-ops only.

**Cause** The first element (after any label) on the source line is not recognized as an instruction or pseudo-op.

**Action** Use only supported instructions or pseudo-ops.

1252-017 The ERRTOK in the args parameter is not valid. Depending upon where you acquired this product, contact either your service representative or your approved supplier.

**Cause** This is an internal error message.

**Action** Contact your service representative or your approved supplier to report the problem.

1252-018 Use a .tc inside a .toc scope only. Precede the .tc statements with a .toc statement.

**Cause** A .tc pseudo-op is only valid after a .toc pseudo-op and prior to a .csect pseudo-op. Otherwise, this message is displayed.

**Action** Ensure that a .toc pseudo-op precedes the .tc pseudo-ops. Any other pseudo-ops should be preceded by a .csect pseudo-op. The .tc pseudo-ops do not have to be followed by a .csect pseudo-op, if they are the last pseudo-ops in a source file.
1252-019 Do not specify externally defined symbols as .byte or .vbyte expression parameters.

**Cause** If the Expression parameter of the .byte or .vbyte pseudo-op contains externally defined symbols (the symbols appear in a .extern or .globl pseudo-op), this message is displayed.

**Action** Verify that the Expression parameter of the .byte or .vbyte pseudo-op does not contain externally defined symbols.

1252-020 Do not specify externally defined symbols as .short Expression parameters.

**Cause** If the Expression parameter of the .short pseudo-op contains externally defined symbols (the symbols appear in an .extern or .globl pseudo-op), this message is displayed.

**Action** Verify that the Expression parameter of the .short pseudo-op does not contain externally defined symbols.

1252-021 The expression must be absolute.

**Cause** The Expression parameter of the .vbyte pseudo-op is not an absolute expression.

**Action** Ensure that the expression is an absolute expression.

1252-022 The first parameter must resolve into an absolute expression from 1 through 4.

**Cause** The first parameter of the .vbyte pseudo-op must be an absolute expression ranging from 1 to 4.

**Action** Verify that the first parameter of the .vbyte pseudo-op resolves to an absolute expression from 1 to 4.

1252-023 The symbol <name> is not defined.

**Cause** An undefined symbol is used in the source program.

**Action** A symbol can be defined as a label, or as the Name parameter of a .csect, .comm, .dsect, .set, .extern, or .globl pseudo-op. The -u flag of the as command suppresses this message.

1252-024 The .stab string must contain a : character.

**Cause** The first parameter of the .stabx pseudo-op is a string constant. It must contain a : (colon). Otherwise, this message is displayed.

**Action** Verify that the first parameter of the .stabx pseudo-op contains a : (colon).

1252-025 The register, base register, or mask parameter is not valid. The register number is limited to the number of registers on your machine.

**Cause** The register number used as the operand of an instruction or pseudo-op is not an absolute value, or the value is out of range of the architecture.

**Action** An absolute expression should be used to specify this value. For PowerPC and POWER family, valid values are in the range of 0-31.

1252-026 Cannot create a temporary file. Check the /tmp directory permissions.

**Cause** This message indicates a permission problem in the /tmp filesystem.

**Action** Check the permissions on the /tmp directory.

1252-027 Warning: Aligning with zeroes: The .short pseudo-op is not on the halfword boundary.

**Cause** This warning indicates that a .short pseudo-op is not on the halfword boundary. The assembler places zeros into the current location until the statement is aligned to a halfword boundary.

**Action** If the user wants to control the alignment, using a .align pseudo-op with the Number parameter set to 1 prior to the .short pseudo-op will perform the same function. A .byte pseudo-op with an Expression parameter set to 0 prior to the .short pseudo-op will perform the same function that the assembler does internally.
1252-028 Cannot reopen the intermediate result file in the /tmp directory. Make sure that the size of the /tmp file system is sufficient to store the file, and check that the file system is not damaged.

**Cause** This message indicates that a system problem occurred while closing the intermediate file and then opening the file again.

**Action** The intermediate file normally resides in the /tmp filesystem. Check the /tmp filesystem space to see if it is large enough to contain the intermediate file.

1252-029 There is not enough memory available now. Cannot allocate the text and data sections. Try again later or use local problem reporting procedures.

**Cause** This is a memory-management problem. It is reported when the `malloc` function is called while allocating the text and data section. There is either not enough main memory, or memory pointers are being corrupted.

**Action** Try again later. If the problem continues to occur, check the applications load for the memory or talk to the system administrator.

1252-030 Cannot create the file `<filename>`. Check path name and permissions.

**Cause** This message indicates that the assembler is unable to create the output file (object file). An object file is created in the specified location if the `-o` flag of the `as` command is used. If the `-o` flag is not used, an object file with the default name of `a.out` is created in the current directory. If there are permission problems for the directory or the path name is invalid, this message is displayed.

**Action** Check the path name and permissions.

1252-031 There is not enough memory available now. Cannot allocate the ESD section. Try again later or use local problem reporting procedures.

**Cause** This is a memory-management problem. It is reported when the `malloc` function is called while allocating the ESD section. There is either not enough main memory, or memory pointers are being corrupted.

**Action** Try again later. If the problem continues to occur, check the applications load for the memory or talk to the system administrator.

1252-032 There is not enough memory available now. Cannot allocate the RLD section. Try again later or use local problem reporting procedures.

**Cause** This is a memory-management problem. It is reported when the `malloc` function is called while allocating the RLD section. There is either not enough main memory, or memory pointers are being corrupted.

**Action** Try again later. If the problem continues to occur, check the applications load for the memory or talk to the system administrator.

1252-033 There is not enough memory available now. Cannot allocate the string section. Try again later or use local problem reporting procedures.

**Cause** This is a memory-management problem. It is reported when the `malloc` function is called while allocating the string section. There is either not enough main memory, or memory pointers are being corrupted.

**Action** Try again later. If the problem continues occur, check applications load for the memory or talk to the system administrator.

1252-034 There is not enough memory available now. Cannot allocate the line number section. Try again later or use local problem reporting procedures.

**Cause** This is a memory-management problem. It is reported when the `malloc` function is called while allocating the line number section. There is either not enough main memory, or memory pointers are being corrupted.

**Action** Try again later. If the problem continues to occur, check the applications load for the memory or talk to the system administrator.

1252-035 through 1252-037 Obsolete messages.
1252-038 Cannot open file <filename>. Check path name and permissions.

**Cause** The specified source file is not found or has no read permission; the listfile or the xcrossfile has no write permission; or the specified path does not exist.

**Action** Check the path name and read/write permissions.

1252-039 Not used currently.

1252-040 The specified expression is not valid. Make sure that all symbols are defined. Check the rules on symbols used in an arithmetic expression concerning relocation.

**Cause** The indicated expression does not resolve into an absolute expression, relocatable expression, external expression, toc relative expression, tocof symbol, or restricted external expression.

**Action** Verify that all symbols are defined. Also, there are some rules concerning relocation on which symbols can be used in an arithmetic expression. See "Expressions" on page 39 for more information.

1252-041 Cannot divide the value by 0 during any arithmetic divisions.

**Cause** During an arithmetic division, the divisor is zero.

**Action** Ensure that the value is not divided by zero.

1252-042 The internal arithmetic operator is not known. Depending upon where you acquired this product, contact either your service representative or your approved supplier.

**Cause** This is an internal error message.

**Action** Contact your service representative or your approved supplier to report the problem.

1252-043 The relocatable assembler expression is not valid. Check that the expressions can be combined.

**Cause** This message is displayed when some invalid arithmetic combinations of the expressions are used.

**Action** Ensure that the correct arithmetic combination is used. See "Expressions" on page 39 for the specific rules of the valid arithmetic combinations for expressions.

1252-044 The specified source character <char> does not have meaning in the command context used.

**Cause** A source character has no meaning in the context in which it is used. For example, .long 3@1, the @ is not an arithmetic operator or an integer digit, and has no meaning in this context.

**Action** Ensure that all characters are valid and have meaning in the context in which they are used.

1252-045 Cannot open the list file <filename>. Check the quality of the file system.

**Cause** This occurs during pass two of the assembler, and indicates a possible filesystem problem or a closing problem with the original listing file.

**Action** Check the file system according to the file path name.

1252-046 Not used currently.

1252-047 There is a nesting underflow. Check for missing .function, .bi or .bb pseudo-ops.

**Cause** This syntax error message is displayed only if debugger pseudo-ops are used. The .function, .bb, and .bi pseudo-ops generate pointers that are saved on a stack with a limiting size of 100 pointers. The .ef, .eb, and .ei pseudo-ops then remove these pointers from the stack. If the number of .ef, .eb, and .ei pseudo-ops encountered is greater than the number of pointers on the stack, this message is displayed.

**Action** Rewrite the code to avoid this problem.

1252-048 Found a symbol type that is not valid when building external symbols. Depending upon where you acquired this product, contact either your service representative or your approved supplier.

**Cause** This is an internal error message.

**Action** Contact your service representative or your approved supplier to report the problem.
There is not enough memory to contain all the hash strings. Depending upon where you acquired this product, contact either your service representative or your approved supplier.

**Cause**  This is an internal error message.

**Action**  Contact your service representative or your approved supplier to report the problem.

There is not enough memory available now. Cannot allocate the debug section. Try again later or use local problem reporting procedures.

**Cause**  This is a memory-management problem. It is reported when the `malloc` function is called while allocating the debug section. There is either not enough main memory, or memory pointers are being corrupted.

**Action**  Try again later. If the problem continues to occur, check the applications load for the memory or talk to the system administrator.

There is an `sclass` type of `Number=<number>` that is not valid. Depending upon where you acquired this product, contact either your service representative or your approved supplier.

**Cause**  This is an internal error message.

**Action**  Contact your service representative or your approved supplier to report the problem.

The specified `.align` parameter must be an absolute value from 0 to 12.

**Cause**  The `Number` parameter of the `.align` pseudo-op is not an absolute value, or the value is not in the range 0-12.

**Action**  Verify that the `Number` parameter resolves into an absolute expression ranging from 0 to 12.

Change the value of the `.org` parameter until it is contained in the current csect.

**Cause**  The value of the parameter for the `.org` pseudo-op causes the location counter to go outside of the current csect.

**Action**  Ensure that the value of the first parameter meets the following criteria:

- Must be a positive value (includes 0).
- Must result in an address that is contained in the current csect.
- Must be an external (E_EXT) or relocatable (E_REL) expression.

The register parameter in `.using` must be absolute and must represent a register on the current machine.

**Cause**  The second parameter of the `.using` pseudo-op does not represent an absolute value, or the value is out of the valid register number range.

**Action**  Ensure that the value is absolute and is within the range of 0-31 for PowerPC and POWER family.

There is a base address in `.using` that is not valid. The base address must be a relocatable expression.

**Cause**  The first parameter of the `.using` pseudo-op is not a relocatable expression.

**Action**  Ensure that the first parameter is relocatable. The first parameter can be a TOC-relative label, a label/name that is relocatable (relocatable=REL), or an external symbol that is defined within the current assembly source as a csect name/TOC entry.

Specify a `.using` argument that references only the beginning of the TOC section. The argument cannot reference locations contained within the TOC section.

**Cause**  The first parameter of the `.using` pseudo-op is a TOC-relative expression, but it does not point to the beginning of the TOC.

**Action**  Verify that the first parameter describes the beginning of the TOC if it is TOC-relative.
1252-057 The external expression is not valid. The symbol cannot be external. If the symbol is external, the symbol must be defined within the assembly using a .toc or a .csect entry.

**Cause**  An external expression other than a csect name or a TOC entry is used for the first parameter of the using pseudo-op.

**Action**  Ensure that the symbol is either not external (not specified by an .extern pseudo-op) or is defined within the assembly source using a TOC entry or csect entry.

1252-058 Warning: The label <name> is aligned with csect <csectname>.

**Cause**  If the label is in the same line of the .csect pseudo-op, this warning is reported when the -w flag of the as command is used. This message indicates that a label may not be aligned as intended. If the label should point to the top of the csect, it should be contained within the csect, in the first line next to the .csect pseudo-op.

**Action**  Evaluate the intent of the label.

1252-059 The register in .drop must be an absolute value that is a valid register number.

**Cause**  The parameter of the .drop pseudo-op is not an absolute value, or the value is not in the range of valid register numbers.

**Action**  Use an absolute value to indicate a valid register. For PowerPC and POWER family, valid register numbers are in the range of 0-31.

1252-060 The register in .drop is not in use. Delete this line or insert a using line previous to this .drop line.

**Cause**  This message indicates that the register represented by the parameter of the .drop pseudo-op was never used in a previous .using statement.

**Action**  Either delete the .drop pseudo-op or insert the .using pseudo-op that should have been used prior to this .drop pseudo-op.

1252-061 A statement within .toc scope is not valid. Use the .tc pseudo-op to define entries within .toc scope.

**Cause**  If a statement other than a .tc pseudo-op is used within the .toc scope, this message is displayed.

**Action**  Place a .tc pseudo-op only inside the .toc scope.

1252-062 The alignment must be a value from 0 to 31.

**Cause**  The optional second parameter (Number) of the .csect parameter defines alignment for the top of the current csect. Alignment must be in the range 0-31. Otherwise, this message is displayed.

**Action**  Ensure that the second parameter is in the valid range.

1252-063 Obsolete message.

1252-064 The .comm size must be an absolute expression.

**Cause**  The second parameter of the .comm pseudo-op must be an absolute expression. Otherwise, this message is displayed.

**Action**  Ensure that the second parameter is an absolute expression.

1252-065 Not used currently.

1252-066 There is not enough memory available now. Cannot allocate the typchk section. Try again later or use local problem reporting procedures.

**Cause**  This is a memory-management problem. It is reported when the malloc function is called while allocating the debug section. There is either not enough main memory, or memory pointers are being corrupted.

**Action**  Try again later. If the problem continues to occur, check the applications load for the memory or talk to the system administrator.

1252-067 The specified common storage class is not valid. Depending upon where you acquired this product, contact either your service representative or your approved supplier.

**Cause**  This is an internal error message.

**Action**  Contact your service representative or your approved supplier to report the problem.
1252-068 The .hash string is set for symbol name already. Check that this is the only .hash statement associated with the symbol name.

**Cause** The Name parameter of the .hash pseudo-op has already been assigned a string value in a previous .hash statement.

**Action** Ensure that the Name parameter is unique for each .hash pseudo-op.

1252-069 The character <char> in the hash string is not valid. The characters in the string must be in the set [0-9A-Fa-f].

**Cause** The characters in the hash string value (the second parameter of the .hash pseudo-op) are required to be in the set [0-9A-Fa-f]. The characters represent a hexadecimal hash code. Otherwise, this message is displayed.

**Action** Ensure that the characters specified by the StringConstant parameter are contained within this set.

1252-070 The specified symbol or symbol type for the hash value is not valid.

**Cause** If the Name parameter for the .hash pseudo-op is not a defined external symbol, this message is displayed.

**Notes:**
1. This message can be suppressed by using the -u flag of the as command.
2. A defined internal symbol (for example, a local label) can also cause this message to be displayed.

**Action** Use the -u flag of the as command, or use the .extern or .globl pseudo-op to define the Name parameter as an external symbol.

1252-071 and 1252-072 Not used currently.

1252-073 There is not enough memory available now. Cannot allocate a segment in memory. Try again later or use local problem reporting procedures.

**Cause** This indicates a malloc realloc, or calloc problem. The following problems can generate this type of error:
- Not enough main memory to allocate
- Corruption in memory pointers
- Corruption in the filesystem

**Action** Check the file systems and memory status.

1252-074 The pseudo-op is not within the text section. The .function, .bf and .ef pseudo-ops must be contained within a csect with one of the following storage classes: RO, PR, XO, SV, DB, GL, TI, or TB.

**Cause** If the .function, .bf and .ef pseudo-ops are not within a csect with a storage mapping class of RO, PR, XO, SV, DB, GL, TI, or TB, this syntax error message is displayed.

**Action** Ensure that the .function, .bf, and .ef pseudo-ops are within the scope of a text csect.

1252-075 The specified number of parameters is not valid.

**Cause** This is a syntax error message. The number of parameters specified with the instruction is incorrect.

**Action** Verify that the correct number of parameters are specified for this instruction.

1252-076 The .line pseudo-op must be contained within a text or data csect.

**Cause** This is a syntax error message. The .line pseudo-op must be within a text or data section. If the .line pseudo-op is contained in a .dsect pseudo-op, or in a .csect pseudo-op with a storage mapping class of BS or UC, this error is displayed.

**Action** Verify that the .line pseudo-op is not contained within the scope of a .dsect; or in a .csect pseudo-op with a storage mapping class of BS or UC.
1252-077  The file table is full. Do not include more than 99 files in any single assembly source file.

**Cause**  The .xline pseudo-op indicates a filename along with the number. These pseudo-ops are generated with the -l option of the m4 command. A maximum of 99 files may be included with this option. If more than 99 files are included, this message is displayed.

**Action**  Ensure that the m4 command has not included more than 99 files in any single assembly source file.

1252-078  The bit mask parameter starting at <positionnumber> is not valid.

**Cause**  This is a syntax error message. In rotate left instructions, there are two input operand formats: rlxx RA,RS,SH,MB,ME, or rlxx RA,RS,SH,BM. This message is displayed only if the second format is used. The BM parameter specifies the mask for this instruction. It must be constructed by certain rules. Otherwise, this message is displayed. See "Extended Mnemonics of 32-bit Fixed-Point Rotate and Shift Instructions" on page 107 for information on constructing the BM parameter.

**Action**  Correct the bit mask value.

1252-079  Found a type that is not valid when counting the RLDs. Depending upon where you acquired this product, contact either your service representative or your approved supplier.

**Cause**  This is an internal error message.

**Action**  Contact your service representative or your approved supplier to report the problem.

1252-080  The specified branch target must be on a full word boundary.

**Cause**  This is a syntax error message. Branch instructions have a target or location to which the program logic should jump. These target addresses must be on a fullword boundary.

**Action**  Ensure that the branch target is on a fullword address (an address that ends in 0, 4, 8, or c). The assembler listing indicates location counter addresses. This is useful when trying to track down this type of problem.

1252-081  The instruction is not aligned properly. The instruction requires machine-specific alignment.

**Cause**  On PowerPC and POWER family, the alignment must be fullword. If this message is displayed, it is probable that an instruction or pseudo-op prior to the current instruction has modified the location counter to result in an address that does not fall on a fullword.

**Action**  Ensure that the instruction is on a fullword address.

1252-082  Use more parameters for the instruction.

**Cause**  Each instruction expects a set number of arguments to be passed to it. If too few arguments are used, this error is displayed.

**Action**  Check the instruction definition to find out how many arguments are needed for this instruction.

1252-083  Use fewer parameters for the instruction.

**Cause**  Each instruction expects a set number of arguments to be passed to it. If too many arguments are used, this error is displayed.

**Action**  Check the instruction definition to find out how many arguments are needed for this instruction.

1252-084 and 1252-085  Obsolete messages.

1252-086  The target of the branch instruction must be a relocatable or external expression.

**Cause**  An absolute expression target is used where a relocatable or external expression is acceptable for a branch instruction.

**Action**  Replace the current branch instruction with an absolute branch instruction, or replace the absolute expression target with a relocatable target.
The target of the branch instruction must be a relocatable or external expression.

**Cause**  
This is a syntax error message. The target of the branch instruction must be either relocatable or external.

**Action**  
Ensure that the target of this branch instruction is either relocatable or external.

Relocatable expressions include label names, `.lcomm` names, `.comm` names, and `.csect` names.

Relocation refers to an entity that represents a memory location whose address or location can and will be changed to reflect run-time locations. Entitles and symbol names that are defined as relocatable or non-relocatable are described in "Expressions" on page 39.

The branch address is out of range. The target address cannot exceed the ability of the instruction to represent the bit size of the branch address value.

**Cause**  
This is a syntax error message. Branch instructions limit the target address sizes to 26 bits, 16 bits, and other instruction-specific sizes. When the target address value cannot be represented in the instruction-specific limiting space, this message is displayed.

**Action**  
Ensure that the target address value does not exceed the instruction's ability to represent the target address (bit size).

The specified displacement is not valid. The instruction displacement must be relocatable, absolute, or external.

**Cause**  
This is a syntax error message. The instruction displacement must be either relocatable; absolute; external which has the XTY_SD or STY_CM symbol type (a csect or common block name); or possibly TOC-relative (but not a negative TOC-relative), depending on the machine platform.

**Action**  
Verify that the displacement is valid for this instruction.

Either the displacement value or the contents of the specified general purpose register, or both, do not yield a valid address.

**Cause**  
Indicates an invalid \(d(r)\) operand. Either \(d\) or \(r\) is missing.

**Action**  
Verify that the base/displacement operand is formed correctly. Correct the programming error, then assemble and link the program again.

**Note:** If \(d\) or \(r\) does not need to be specified, 0 should be put in the place.

The specified instruction is not supported by this machine.

**Cause**  
This is an internal error message.

**Action**  
The `<parm #>` parameter must be absolute.

The indicated parameter must be absolute (nonrelocatable, nonexternal).

**Action**  
Refer to the specific instruction article for the instruction syntax.

Obsolete messages.
The parameter <parm #> must be within range for the specific instruction.

**Cause**  
This error occurs in the following situations:

- The parameter value does not lie within the lower and upper bounds.
- The parameter value for the SPR encoding is undefined.
- The parameter value for the rotate and shift instructions is beyond the limitation.

**Action**  
See the specific instruction article for the instruction definition. See "Extended Mnemonics of Moving from or to Special-Purpose Registers" on page 102 for the list of SPR encodings. In general, if the assembly mode is **com**, **pwr**, or **pwr2**, the SPR range is 0 to 31. Otherwise, the SPR range is 0 to 1023. See "Extended Mnemonics of 32-bit Fixed-Point Rotate and Shift Instructions" on page 107 for information on restrictions. Change the source code, then assemble and link the program again.

**Warning:** The alignment for label <name> is not valid. The label requires machine-specific alignment.

**Cause**  
Indicates that a label is not aligned properly to be the subject of a branch. In other words, the label is not aligned to a fullword address (an address ending in 0, 4, 8, or c).

**Action**  
To control the alignment, a .align pseudo-op prior to the label will perform the alignment function. Also, a .byte pseudo-op with a parameter of 0 or a .short pseudo-op with a parameter of 0 prior to the label will shift the alignment of the label.

**Warning:** Aligning with zeros: The .long pseudo-op is not on fullword boundary.

**Cause**  
Indicates that a .long pseudo-op exists that is not aligned properly on a fullword internal address (an address that ends in 0, 4, 8, or c). The assembler generates zeros to properly align the statement.

**Action**  
To control the alignment, a .align pseudo-op with a parameter of 2 prior to the .long pseudo-op will perform the alignment. Also, a .byte pseudo-op with a parameter of 0 or a .short pseudo-op with a parameter of 0 prior to the .long pseudo-op will perform the alignment.

**Warning:** Aligning with zeros in program csect.

**Cause**  
If the .align pseudo-op is used within a .csect of type [PR] or [GL], and the .align pseudo-op is not on a fullword address (for PowerPC and POWER family, all instructions are four bytes long and are fullword aligned), the assembler performs alignment by padding zeros, and this warning message is displayed. It is also displayed when a fullword alignment occurs in other pseudo-op statements.

**Action**  
Look for a reason why the alignment is not on a fullword. This could indicate a possible pseudo-op or instruction in the wrong place.

**Warning:** Csect alignment has changed. To change alignment, check previous .csect statements.

**Cause**  
The beginning of the csect is aligned according to a default value (2, fullword) or the Number parameter. This warning indicates that the alignment that was in effect when the csect was created has been changed later in the source code.

The csect alignment change can be caused by any of the following:

- The Number parameter of the .csect pseudo-op specifies a value greater than previous .csect pseudo-ops that have the same Qualname.
- The Number parameter of a .align pseudo-op specifies a value greater than the current csect alignment.
- A .double pseudo-op is used, which causes the alignment to increase to 3. If the current csect alignment is less than 3, this warning is reported.

**Action**  
This message may or may not indicate a problem, depending on the user’s intent. Evaluate whether a problem has occurred or not.
Warning: The <inst. format> instruction is not supported by this machine.

**Cause** This is an internal error message.

**Action** Contact your service representative or your approved supplier to report the problem.

**Obsolete messages.**

The sort failed with status <number>. Check the condition of the system sort command or use local problem reporting procedures.

**Cause** When the -x flag of the as command is used from the command line, the system sort routine is called. If this call is not successful, this message is displayed. Either the sort utility is not available, or a system problem has occurred.

**Action** Check the condition of the system sort command, check the system itself (using the fsck command), or use local problem reporting procedures.

There is a system error from <name>. Check the condition of the system sort command or use local problem reporting procedures.

**Cause** name has the sort command. When the -x flag of the as command is used from the command line, the system sort routine is called. The assembler forks a process to call the sort utility. If this fork fails to exec the sort routine, this message is displayed. Either the sort utility is not available, or a system problem has occurred.

**Action** Check the condition of the system sort command, check the system itself (using the fsck command), or use local problem reporting procedures.

"Assembler:"

**Cause** This line defines a header to the standard error output to indicate that it is an assembly program.

"line <number>"

**Cause** number contains the line number on which an error or warning resides. When assembling a source program, this message is displayed prior to the error/warning message on the screen. This message is also printed prior to the error/warning message in the assembler listing file.

".xref"

**Cause** This message defines the default suffix extension for the file name of the symbol cross-reference file.

".lst"

**Cause** This message defines the default suffix extension for the file name of the assembler listing file.

"SYMBOL FILE CSEC LINENO"

**Cause** This line defines the heading of the symbol cross-reference file.

Define several formats used in the assembler listing file.

Obsolete, replaced by 1252-179.

Define the spaces or formats for the assembler listing file.

Define formats for output numbers and names.

Defines 8 spaces that are used in the listing file.

Defines a format used in the listing file.

Formats for output of a number.
There is an error in the collect pointer. Use local problem reporting procedures.

**Cause**  This is an internal error message.

**Action**  Contact your service representative or your approved supplier to report the problem.

Syntactic error

**Cause**  If an error occurred in the assembly processing and the error is not defined in the message catalog, this generic error message is used. This message covers both pseudo-ops and instructions. Therefore, a usage statement would be useless.

**Action**  Determine intent and source line construction, then consult the specific instruction article to correct the source line.

The `.function` Size must be an absolute expression.

**Cause**  The Size parameter of the `.function` pseudo-op represents the size of the function. It must be an absolute expression.

**Action**  Change the Size parameter, then assemble and link the program again.

Warning: Any initialized data in `<name>` csect of BS or UC storage class is ignored but required to establish length.

**Cause**  Indicates that the statements in the csect with a storage mapping class of BS or UC are used to calculate length of the csect and are not used to initialize data.

**Action**  None.

Obsolete, replaced by 1252-180 and 1252-181.

Invalid `.machine` assembly mode operand: `<name>`

**Cause**  The `.machine` pseudo-op is used in a source program to indicate the assembly mode value. This message indicates that an undefined value was used.

**Action**  See the `.machine Pseudo-op` on page 490 for a list of the defined assembly mode values.

Invalid `.source` language identifier operand: `<name>`

**Cause**  The `.source` pseudo-op indicates the source language type (C, FORTRAN, etc.). This message indicates that an invalid source language type was used.

**Action**  See the `.source` pseudo-op for a list of the defined language types.

Instruction `<name1>` is not implemented in the current assembly mode `<name2>`.

**Cause**  Instructions that are not in the POWER family/PowerPC intersection area are implemented only in certain assembly modes. This message indicates that the instruction in the source program is not supported in the indicated assembly mode.

**Action**  Use a different assembly mode or a different instruction.

The first operand value of value is not valid for PowerPC. A BO field of 6, 7 14, 15, or greater than 20 is not valid.

**Cause**  In branch conditional instructions, the first operand is the BO field. If the input value is outside of the required values, this message is displayed.

**Action**  See the “Features of the AIX Assembler” on page 1 for the BO field encoding information to find the correct value of the input operand.

This instruction form is not valid for PowerPC. The register used in operand two must not be zero and must not be the same as the register used in operand one.

**Cause**  In the update form of fixed-point load instructions, PowerPC requires that the RA operand not be equal to zero and that it not be equal to RT. If these requirements are violated, this message is displayed.

**Action**  See the “Features of the AIX Assembler” on page 1 for a list of these instructions, and refer to the instruction articles for the syntax and restrictions of these instructions. Change the source code, then assemble and link the program again.
Internal error related to the source program domain. Depending upon where you acquired this product, contact your service representative or your approved supplier.

**Cause**
This is an internal error message.

**Action**
Contact your service representative or your approved supplier to report the problem.

**Warning:** Instruction `<name>` functions differently between PowerPC and POWER™.

**Cause**
This warning message is not displayed unless the `lw` flag of the `as` command is used in the command line. Some instructions have the same op code in PowerPC and POWER, but are functionally different. This message provides a warning if the assembly mode is `com` and these instructions are used.

**Action**
See “Functional Differences for POWER family and PowerPC Instructions” on page 114 for information on instructions that have the same op code but are functionally different in POWER and PowerPC.

The second operand is not valid. For 32-bit implementation, the second operand must have a value of zero.

**Cause**
In the fixed-point compare instructions, the value in the L field must be zero for 32-bit implementation. Also, if the `mtsri` instruction is used in one of the PowerPC assembly modes, the RA operand must contain zero. Otherwise, this message is displayed.

**Action**
Put the correct value in the second operand, then assemble and link the program again.

Displacement must be divisible by 4.

**Cause**
If an instruction has the DS form, its 16-bit signed displacement value must be divisible by 4. Otherwise, this message is displayed.

**Action**
Change the displacement value, then assemble and link the program again.

The sum of argument 3 and 4 must be less than 33.

**Cause**
When some extended mnemonics for word rotate and shift instructions are converted to the base instruction, the values of the third and fourth operands are added to calculate the SH field, MB field, or ME field. Since these fields are 5 bits in length, the sum of the third and fourth operands must not be greater than 32.

**Action**
See “Extended Mnemonics of 32-bit Fixed-Point Rotate and Shift Instructions” on page 107 for information on converting the extended mnemonic to the base instruction. Change the value of the input operands accordingly, then assemble and link the program again.

The value of operand 3 must be greater than or equal to the value of operand 4.

**Cause**
When some extended mnemonics for word rotate and shift instructions are converted to the base instruction, the value of the fourth operand is subtracted from the value of the third operand to get the ME or MB field. The result must be positive. Otherwise, this message is displayed.

**Action**
See “Extended Mnemonics of 32-bit Fixed-Point Rotate and Shift Instructions” on page 107 for information on converting the extended mnemonic to the base instruction. Change the value of the input operands accordingly, then assemble and link the program again.

Warning: Special-purpose register number 6 is used to designate the DEC register when the assembly mode is `name`.

**Cause**
This warning is displayed when the `mfdec` instruction is used and the assembly mode is `any`. The DEC encoding for the `mfdec` instruction is 22 for PowerPC and 6 for POWER. When the assembly mode is `any`, the POWER encoding number is used to generate the object code, and this message is displayed to indicate this.

**Action**
None.
The d(r) format is not valid for operand <value>.

**Cause**
Indicates an assembly programming error. The d(r) format is used in the place that a register number or an immediate value is required.

**Action**
Correct the programming error, then assemble and link the program again.

Warning: A hash code value should be 10 bytes long.

**Cause**
When the [hash] pseudo-op is used, the second parameter, StringConstant, gives the actual hash code value. This value should contain a 2-byte language ID, a 4-byte general hash, and a 4-byte language hash. The hash code value should be 10 bytes long. If the value length is not 10 bytes and the -w flag of the as command is used, this warning is displayed.

**Action**
Use the correct hash code value.

A system problem occurred while processing file <filename>.

**Cause**
A problem with system I/O developed dynamically. This message is produced by the assembler to indicate a [write, putc, fclose] error. The I/O problem could be caused by corruption of the filesystem or not enough space in the file systems.

**Action**
Check the proper file system according to the path name reported.

Invalid -m flag assembly mode operand: <name>.

**Cause**
When an invalid assembly mode is entered on the command line using -m flag of the as command, this message is displayed.

**Action**
See the Chapter 5, “Assembling and Linking a Program,” on page 53 for the defined assembly modes.

The first operand's value <value> is not valid for PowerPC. The third bit of the BO field must be one for the Branch Conditional to Count Register instruction.

**Cause**
If the third bit of the BO operand is zero for the “bcctr or bcc (Branch Conditional to Count Register) Instruction” on page 147, the instruction form is invalid and this message is displayed.

**Action**
Change the third bit to one, then assemble and link the program again.

This instruction form is not valid for PowerPC. RA, and RB if present in the instruction, cannot be in the range of registers to be loaded. Also, RA=RT=0 is not allowed.

**Cause**
In multiple register load instructions, PowerPC requires that the RA operand, and the RB operand if present in the instruction format, not be in the range of registers to be loaded. Also RA=RT=0 is not allowed. Otherwise, this message is displayed.

**Action**
Check the register number of the RA, RB, or RT operand to ensure that this requirement is met.

The value of the first operand must be zero for PowerPC.

**Cause**
If the POWER svca instruction is used in one of the PowerPC assembly modes, the first operand is the SV operand. This operand must be zero. Otherwise, this message is displayed.

**Action**
Put zero into the first operand, or use the PowerPC sc instruction, which does not require an operand.

This instruction form is not valid for PowerPC. The register used in operand two must not be zero.

**Cause**
For the update form of fixed-point store instructions and floating-point load and store instructions, PowerPC requires that the RA operand not be equal to zero. Otherwise, this message is displayed.

**Action**
Check the register number specified by the RA operand, then assemble and link the source code again.
Specify a name with the -<flagname> flag.

**Cause** The -n and -o flags of the as command require a filename as a parameter. The -m flag of the as command requires a mode name as a parameter. If the required name is missing, this error message is displayed. This message replaces message 1252-035.

**Action** Provide a filename with the -n and -o flags of the as command, and provide a mode name with the -m flag of the as command.

-<name> is not a recognized flag.

**Cause** An undefined flag was used on the command line. This message replaces message 1252-036.

**Action** Make a correction and run the command again.

Only one input file is allowed.

**Cause** More than one input source file was specified on the command line. This message replaces message 1252-037

**Action** Specify only one input source file at a time.

The Assembler command has the following syntax:

```
as [-l(ListFile) -s(ListFile) -n Name -o ObjectFile [-w -W] -x(XCrossFile) -u -m ModeName [InputFile]]
```

**Cause** This message displays the usage of the as command.

**Action** None.

The displacement must be greater than or equal to <value1> and less than or equal to <value2>.

**Cause** For 16-bit displacements, the limits are 32767 and -32768. If the displacement is out of range, this message is displayed. This message replaces message 1252-106.

**Action** See the specific instruction articles for displacement requirements.

The .extern symbol is not valid. Check that the .extern Name is a relocatable expression.

**Cause** The Name parameter of the .extern pseudo-op must specify a relocatable expression. This message is displayed if the Name parameter of the .extern pseudo-op does not specify a relocatable expression. For information on relocatable and nonrelocatable expressions, see message 1252-004.

**Action** Ensure that the Name parameter of the .extern pseudo-op is a relocatable expression.

Warning: The immediate value for instruction <name> is <value>. It may not be portable to a 64-bit machine if this value is to be treated as an unsigned value.

**Cause** This warning is reported only for the addis instruction (or the lis extended mnemonic of the addis instruction). The immediate value field of these instructions is defined as a signed integer, which should have a valid value range of -32768 to 32767. To maintain compatibility with the cau instruction, however, this range is expanded to -65536 to 65535. This should cause no problems in a 32-bit mode, because there is nowhere for sign extension to go. However, this will cause a problem on a 64-bit machine, because sign extension propagates across the upper 32 bits of the register.

**Action** Use caution when using the addis instruction to construct an unsigned integer. The addis instruction has different semantics on a 32-bit implementation (or in 32-bit mode on a 64-bit implementation) than it does in 64-bit mode. The addis instruction with an unsigned integer in 32-bit mode cannot be directly ported to a 64-bit mode. The code sequence to construct an unsigned integer in 64-bit mode is significantly different from that needed in 32-bit mode.
Too many .machine "push" instructions without corresponding .machine "pop" instructions.

**Cause**  The maximum size of the assembly stack has been exceeded. More than 100 entries have been added to the stack with .machine "push" but not removed with .machine "pop".

**Action**  Change the source program to eliminate the assembly stack overflow condition.

A .machine "pop" is seen without a matching .machine "push".

**Cause**  Pseudo-op .machine "pop" attempted to remove an entry from the assembly stack, but the stack is empty. The source program may be missing a .machine "push".

**Action**  Correct the source program.

The .ref pseudo-op cannot appear in section <name>.

**Cause**  A .ref pseudo-op appears in a dsect or a csect with a storage mapping class of BS or UC, which is not permitted.

**Action**  Change the source program.

The operand of the .ref <name> is not a relocatable symbol.

**Cause**  .ref pseudo-op operand name is one of the following items: a dsect name or label, a csect name or label with a storage mapping class of BS or UC, a .set operand which represents an item that is not relocatable, or a constant value.

**Action**  Correct the source program.

The maximum number of sections or symbols that an expression can refer to has been exceeded.

**Cause**  An expression refers to more than 50 control sections (csects or dsects).

**Action**  Correct the source program.

File# Line# Mode Name Loc Ctr Object Code Source

**Cause**  This line defines the heading of the assembler listing file without the mnemonics cross reference of POWER and PowerPC.

File# Line# Mode Name Loc Ctr Object Code PowerPC Source

**Cause**  This is one of the headings of the assembler listing file with the mnemonics cross-reference of POWER and PowerPC. The assembler listing column labeled PowerPC contains PowerPC mnemonics for statements where the source program uses POWER mnemonics. This message is used for assembly modes of the PowerPC category (including com, ppc, 601, and any).

File# Line# Mode Name Loc Ctr Object Code POWER Source

**Cause**  This is one of the headings of the assembler listing file with the mnemonics cross-reference of POWER and PowerPC. The assembler listing column labeled POWER contains POWER mnemonics for statements where the source program uses PowerPC mnemonics. This message is used for assembly modes of the POWER category (including pwr and pwr2).

Storage mapping class <name> is not valid for .comm pseudo-op. RW is used as the storage mapping class for the object code.

**Cause**  The storage mapping class of the .comm pseudo-op is some value other than the valid values (TD, RW, BS, and UC). The assembler reports this as a warning and uses RW as the storage mapping class.

**Action**  Change the source program.

TD csect only allowed inside ".toc" scope.

**Cause**  A csect with storage mapping class TD has been used without first using the .toc pseudo-op.

**Action**  Use the .toc pseudo-op before this instruction.
TOC anchor must be defined to use a TOC-relative reference to <name>. Include a .toc pseudo-op in the source.

**Cause** A TOC-relative reference is being used, but the TOC anchor is not defined. This can happen if an external TD symbol is defined and used as a displacement in a D-form instruction, but there is no .toc pseudo-op in the source program.

**Action** Use the .toc pseudo-op in the program.

Warning: Operand is missing from pseudo-op.

**Cause** An operand required for pseudo-ops .byte, .vbyte, .short, .long, or .llong is missing.

**Action** Provide an initial value for the data storage area created by these pseudo-ops.

Warning: The maximum length of a stabstring is <number> characters. Extra characters have been discarded.

**Cause** A stabstring is limited in length; the specified stabstring is greater than the maximum length of a single string.

**Action** Split the string into 2 or more strings, continuing the information from one stabstring to the next.

Warning: The alignment of the current csect is less than the alignment specified with the .align pseudo-op.

**Cause** The alignment of the csect is not as strict as the alignment required by the use of a .align pseudo-op within that csect.

**Action** The .align pseudo-op specifies alignment of an item within the csect; the alignment specified for the csect should be equal to or greater than this value. For example, if the csect requires word alignment, and a .llong within the csect requires double-word alignment, there is a potential for the .llong value to ultimately (after linking) be only word-aligned. This may not be what is intended by the user.

Zero is used in the L operand for the <instruction> instruction.

**Cause** Some compare instructions allowed the L operand to be optional in 32-bit mode. In 64-bit mode, the operand is not optional.

**Action** All 4 operands should be specified for the instruction, or, alternatively, use an extended mnemonic.

Invalid value for environment variable OBJECT_MODE. Set the OBJECT_MODE environment variable to 32 or 64 or use the -a32 or -a64 option.

**Cause** The value of the OBJECT_MODE environment variable is not recognized by the assembler.

**Action** Set the OBJECT_MODE environment variable to either 32 or 64, or use the -a32 or -a64 command line option. Any other value for the environment variable has no meaning to the assembler.

Invalid reference to label <name>: .function pseudo-op must refer to a csect.

**Cause** The .function pseudo-op referred to a local label.

**Action** The reference <name> should be the name (label) of a csect.

Only <name> should be used for relocatable expressions.

**Cause** The expression used to initialize <name> contains references to externally defined symbols (i.e. the symbols appear in .extern pseudo-op).

**Action** Verify that no externally defined symbols are contained within the expression operands for <name>. Relocation in 32-bit mode can only be applied to 32-bit quantities; in 64-bit mode relocation can only be applied to 64-bit quantities.
Assembly mode is not specified. Set the OBJECT_MODE environment variable to 32 or 64 or use the -a32 or -a64 option.

**Cause**  The environment variable contains the value 32_64.

**Action**  Set the OBJECT_MODE environment variable to either 32 or 64, or use the -a32 or -a64 command line option.

Values specified with the .set pseudo-op are treated as 32-bit signed numbers. Unexpected results may occur when these values are used in a .llong expression.

**Cause**  In 32-bit mode, an expression that results from the use of .set has been used to set the initial value of a .llong.

**Action**  For initializing .llong's when in 32-bit mode, values are treated as 64-bit. If a .set symbol whose most significant bit is set is set is used as part of the initialization, the value may not be interpreted in a manner intended by the user. For example, the value 0xFFFF_0000 may have been intended to be a positive 64-bit quantity, but is a negative 32-bit number which would be sign extended to become 0xFFFF_FFFF_FFFF_0000.

**Warning:** The immediate value for instruction `<instruction>` is `<number>`. It may not be portable to a 64-bit machine if this value is to be treated as an unsigned value.

**Cause**  This is a alternate version of message 173; see above for more information.
Appendix B. Instruction Set Sorted by Mnemonic

In the Instruction Set Sorted by Mnemonic table the Implementation column contains the following information:

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com</td>
<td>Supported by POWER family, POWER2, and PowerPC implementations.</td>
</tr>
<tr>
<td>POWER family</td>
<td>Supported only by POWER family and POWER2 implementations.</td>
</tr>
<tr>
<td>POWER2</td>
<td>Supported only by POWER2 implementations.</td>
</tr>
<tr>
<td>PowerPC</td>
<td>Supported only by PowerPC architecture.</td>
</tr>
<tr>
<td>PPC opt.</td>
<td>Defined only in PowerPC architecture and is an optional instruction.</td>
</tr>
<tr>
<td>603 only</td>
<td>Supported only on the PowerPC 603 RISC Microprocessor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a[o][.]</td>
<td>Add Carrying</td>
<td>POWER family</td>
<td>XO</td>
<td>31</td>
<td>10</td>
</tr>
<tr>
<td>abs[o][.]</td>
<td>Absolute</td>
<td>POWER family</td>
<td>XO</td>
<td>31</td>
<td>360</td>
</tr>
<tr>
<td>add[o][.]</td>
<td>Add</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>266</td>
</tr>
<tr>
<td>addc[o][.]</td>
<td>Add Carrying</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>10</td>
</tr>
<tr>
<td>adde[o][.]</td>
<td>Add Extended</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>138</td>
</tr>
<tr>
<td>addi</td>
<td>Add Immediate</td>
<td>PowerPC</td>
<td>D</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>addic</td>
<td>Add Immediate Carrying</td>
<td>PowerPC</td>
<td>D</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>addic.</td>
<td>Add Immediate Carrying and Record</td>
<td>PowerPC</td>
<td>D</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>addis</td>
<td>Add Immediate Shifted</td>
<td>PowerPC</td>
<td>D</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>addme[o][.]</td>
<td>Add to Minus One Extended</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>234</td>
</tr>
<tr>
<td>addze[o][.]</td>
<td>Add to Zero Extended</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>202</td>
</tr>
<tr>
<td>ae[o][.]</td>
<td>Add Extended</td>
<td>POWER family</td>
<td>XO</td>
<td>31</td>
<td>138</td>
</tr>
<tr>
<td>ai</td>
<td>Add Immediate</td>
<td>POWER family</td>
<td>D</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>ai.</td>
<td>Add Immediate and Record</td>
<td>POWER family</td>
<td>D</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>ame[o][.]</td>
<td>Add to Minus One Extended</td>
<td>POWER family</td>
<td>XO</td>
<td>31</td>
<td>234</td>
</tr>
<tr>
<td>and[.]</td>
<td>AND</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>28</td>
</tr>
<tr>
<td>andc[.]</td>
<td>AND with Complement</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>60</td>
</tr>
<tr>
<td>andi.</td>
<td>AND Immediate</td>
<td>PowerPC</td>
<td>D</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>andil.</td>
<td>AND Immediate Lower</td>
<td>POWER family</td>
<td>D</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>andis.</td>
<td>AND Immediate Shifted</td>
<td>PowerPC</td>
<td>D</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Purpose</td>
<td>Family</td>
<td>Register</td>
<td>Width</td>
<td>Size</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>--------</td>
<td>----------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>andiu</td>
<td>AND Immediate Upper</td>
<td>POWER family</td>
<td>D</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>aze[o][]</td>
<td>Add to Zero Extended</td>
<td>POWER family</td>
<td>XO</td>
<td>31</td>
<td>202</td>
</tr>
<tr>
<td>b[l][a]</td>
<td>Branch</td>
<td>com</td>
<td>I</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>bc[l][a]</td>
<td>Branch Conditional</td>
<td>com</td>
<td>B</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>bcc[l]</td>
<td>Branch Conditional to Count Register</td>
<td>POWER family</td>
<td>XL</td>
<td>19</td>
<td>528</td>
</tr>
<tr>
<td>bcctr[l]</td>
<td>Branch Conditional to Count Register</td>
<td>PowerPC</td>
<td>XL</td>
<td>19</td>
<td>528</td>
</tr>
<tr>
<td>bclr[l]</td>
<td>Branch Conditional Link Register</td>
<td>PowerPC</td>
<td>XL</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td>bcr[l]</td>
<td>Branch Conditional Register</td>
<td>POWER family</td>
<td>XL</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td>cal</td>
<td>Compute Address Lower</td>
<td>POWER family</td>
<td>D</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>cau</td>
<td>Compute Address Upper</td>
<td>POWER family</td>
<td>D</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>cax[o][]</td>
<td>Compute Address</td>
<td>POWER family</td>
<td>XO</td>
<td>31</td>
<td>266</td>
</tr>
<tr>
<td>clcs</td>
<td>Cache Line Compute Size</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>531</td>
</tr>
<tr>
<td>clf</td>
<td>Cache Line Flush</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>118</td>
</tr>
<tr>
<td>cli</td>
<td>Cache Line Invalidate</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>502</td>
</tr>
<tr>
<td>cmp</td>
<td>Compare</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>cmpi</td>
<td>Compare Immediate</td>
<td>com</td>
<td>D</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>cmpl</td>
<td>Compare Logical</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>cmpli</td>
<td>Compare Logical Immediate</td>
<td>com</td>
<td>D</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>cntlz[.]</td>
<td>Count Leading Zeros</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>26</td>
</tr>
<tr>
<td>cntlzw[.]</td>
<td>Count Leading Zeros Word</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>26</td>
</tr>
<tr>
<td>crand</td>
<td>Condition Register AND</td>
<td>com</td>
<td>XL</td>
<td>19</td>
<td>257</td>
</tr>
<tr>
<td>crandc</td>
<td>Condition Register AND with Complement</td>
<td>com</td>
<td>XL</td>
<td>19</td>
<td>129</td>
</tr>
<tr>
<td>creqv</td>
<td>Condition Register Equivalent</td>
<td>com</td>
<td>XL</td>
<td>19</td>
<td>289</td>
</tr>
<tr>
<td>crnand</td>
<td>Condition Register NAND</td>
<td>com</td>
<td>XL</td>
<td>19</td>
<td>225</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Description</td>
<td>Family</td>
<td>Type</td>
<td>Offset</td>
<td>Cycle</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------</td>
<td>-----------------</td>
<td>------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>crnor</td>
<td>Condition Register NOR</td>
<td>com</td>
<td>XL</td>
<td>19</td>
<td>33</td>
</tr>
<tr>
<td>cror</td>
<td>Condition Register OR</td>
<td>com</td>
<td>XL</td>
<td>19</td>
<td>449</td>
</tr>
<tr>
<td>crorc</td>
<td>Condition Register OR with Complement</td>
<td>com</td>
<td>XL</td>
<td>19</td>
<td>417</td>
</tr>
<tr>
<td>cxor</td>
<td>Condition Register XOR</td>
<td>com</td>
<td>XL</td>
<td>19</td>
<td>193</td>
</tr>
<tr>
<td>dcbf</td>
<td>Data Cache Block Flush</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>86</td>
</tr>
<tr>
<td>dcbi</td>
<td>Data Cache Block Invalidate</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>470</td>
</tr>
<tr>
<td>dcbst</td>
<td>Data Cache Block Store</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>54</td>
</tr>
<tr>
<td>dcbt</td>
<td>Data Cache Block Touch</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>278</td>
</tr>
<tr>
<td>dcbtst</td>
<td>Data Cache Block Touch for Store</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>246</td>
</tr>
<tr>
<td>dcbz</td>
<td>Data Cache Block Set to Zero</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>1014</td>
</tr>
<tr>
<td>dclst</td>
<td>Data Cache Line Store</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>630</td>
</tr>
<tr>
<td>dclz</td>
<td>Data Cache Line Set to Zero</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>1014</td>
</tr>
<tr>
<td>dcs</td>
<td>Data Cache Synchronize</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>598</td>
</tr>
<tr>
<td>div[o][]</td>
<td>Divide</td>
<td>POWER family</td>
<td>XO</td>
<td>31</td>
<td>331</td>
</tr>
<tr>
<td>divs[o][]</td>
<td>Divide Short</td>
<td>POWER family</td>
<td>XO</td>
<td>31</td>
<td>363</td>
</tr>
<tr>
<td>divw[o][]</td>
<td>Divide Word</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>491</td>
</tr>
<tr>
<td>divwu[o][]</td>
<td>Divide Word Unsigned</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>459</td>
</tr>
<tr>
<td>doz[o][]</td>
<td>Difference or Zero</td>
<td>POWER family</td>
<td>XO</td>
<td>31</td>
<td>264</td>
</tr>
<tr>
<td>dozi</td>
<td>Difference or Zero Immediate</td>
<td>POWER family</td>
<td>D</td>
<td>09</td>
<td></td>
</tr>
<tr>
<td>eciwx</td>
<td>External Control in Word Indexed</td>
<td>PPC opt.</td>
<td>X</td>
<td>31</td>
<td>310</td>
</tr>
<tr>
<td>ecowx</td>
<td>External Control out Word Indexed</td>
<td>PPC opt.</td>
<td>X</td>
<td>31</td>
<td>438</td>
</tr>
<tr>
<td>eieio</td>
<td>Enforce In-order Execution of I/O</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>854</td>
</tr>
<tr>
<td>eqv[.]</td>
<td>Equivalent</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>284</td>
</tr>
<tr>
<td>exts[.]</td>
<td>Extend Sign</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>922</td>
</tr>
<tr>
<td>extsb[.]</td>
<td>Extend Sign Byte</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>954</td>
</tr>
<tr>
<td>extsh[.]</td>
<td>Extend Sign Halfword</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>922</td>
</tr>
<tr>
<td>fa[.]</td>
<td>Floating Add</td>
<td>POWER family</td>
<td>A</td>
<td>63</td>
<td>21</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Architecture</td>
<td>Register Width</td>
<td>Group Width</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>---------------</td>
<td>----------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>fabs[]</td>
<td>Floating Absolute Value</td>
<td>com</td>
<td>X</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fadd[]</td>
<td>Floating Add</td>
<td>PowerPC</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fadds[]</td>
<td>Floating Add Single</td>
<td>PowerPC</td>
<td>A</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>fcir[]</td>
<td>Floating Convert to Integer Word</td>
<td>POWER family</td>
<td>X</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fcirz[]</td>
<td>Floating Convert to Integer Word with Round to Zero</td>
<td>POWER family</td>
<td>X</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fcmpo</td>
<td>Floating Compare Ordered</td>
<td>com</td>
<td>X</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fcmpu</td>
<td>Floating Compare Unordered</td>
<td>com</td>
<td>XL</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fctiw[]</td>
<td>Floating Convert to Integer Word</td>
<td>PowerPC</td>
<td>X</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fctiwz[]</td>
<td>Floating Convert to Integer Word with Round to Zero</td>
<td>PowerPC</td>
<td>XL</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fd[]</td>
<td>Floating Divide</td>
<td>POWER family</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fdiv[]</td>
<td>Floating Divide</td>
<td>PowerPC</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fddiv[]</td>
<td>Floating Divide Single</td>
<td>PowerPC</td>
<td>A</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>fm[]</td>
<td>Floating Multiply</td>
<td>POWER family</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fma[]</td>
<td>Floating Multiply-Add</td>
<td>POWER family</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fmadd[]</td>
<td>Floating Multiply-Add</td>
<td>PowerPC</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fmadds[]</td>
<td>Floating Multiply-Add Single</td>
<td>PowerPC</td>
<td>A</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>fmr[]</td>
<td>Floating Move Register</td>
<td>com</td>
<td>X</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fms[]</td>
<td>Floating Multiply-Subtract</td>
<td>POWER family</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fmsub[]</td>
<td>Floating Multiply-Subtract</td>
<td>PowerPC</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fmsubs[]</td>
<td>Floating Multiply-Subtract Single</td>
<td>PowerPC</td>
<td>A</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>fmul[]</td>
<td>Floating Multiply</td>
<td>PowerPC</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fmuls[]</td>
<td>Floating Multiply Single</td>
<td>PowerPC</td>
<td>A</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>fnabs[]</td>
<td>Floating Negative Absolute Value</td>
<td>com</td>
<td>X</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fneg[]</td>
<td>Floating Negate</td>
<td>com</td>
<td>X</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Description</td>
<td>Family</td>
<td>Length</td>
<td>Offset</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>fnma[]</td>
<td>Floating Negative Multiply-Add</td>
<td>POWER family</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fnmadd[]</td>
<td>Floating Negative Multiply-Add</td>
<td>PowerPC</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fnmadds[]</td>
<td>Floating Negative Multiply-Add Single</td>
<td>PowerPC</td>
<td>A</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>fnms[]</td>
<td>Floating Negative Multiply-Subtract</td>
<td>POWER family</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fnmsub[]</td>
<td>Floating Negative Multiply-Subtract</td>
<td>PowerPC</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fnmsubs[]</td>
<td>Floating Negative Multiply-Subtract Single</td>
<td>PowerPC</td>
<td>A</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>fres[]</td>
<td>Floating Reciprocal Estimate Single</td>
<td>PPC opt.</td>
<td>A</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>frsp[]</td>
<td>Floating Round to Single Precision</td>
<td>com</td>
<td>X</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>frsqrte[]</td>
<td>Floating Reciprocal Square Root Estimate</td>
<td>PPC opt.</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fs[]</td>
<td>Floating Subtract</td>
<td>POWER family</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fse[]</td>
<td>Floating-Point Select</td>
<td>PPC opt.</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fsqrt[]</td>
<td>Floating Square Root</td>
<td>POWER2</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fsub[]</td>
<td>Floating Subtract</td>
<td>PowerPC</td>
<td>A</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>fsubs[]</td>
<td>Floating Subtract Single</td>
<td>PowerPC</td>
<td>A</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>icbi</td>
<td>Instruction Cache Block Invalidate</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>ics</td>
<td>Instruction Cache Synchronize</td>
<td>POWER family</td>
<td>X</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>isync</td>
<td>Instruction Synchronize</td>
<td>PowerPC</td>
<td>X</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>Load</td>
<td>POWER family</td>
<td>D</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>lbrix</td>
<td>Load Byte-Reversed Indexed</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>lbz</td>
<td>Load Byte and Zero</td>
<td>com</td>
<td>D</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>lbzu</td>
<td>Load Byte and Zero with Update</td>
<td>com</td>
<td>D</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>lbzux</td>
<td>Load Byte and Zero with Update Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>lbzx</td>
<td>Load Byte and Zero Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Com</td>
<td>Register</td>
<td>Opcode</td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------</td>
<td>-----</td>
<td>----------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>lfd</td>
<td>Load Floating-Point Double</td>
<td>com</td>
<td>D</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Ifdu</td>
<td>Load Floating-Point Double with Update</td>
<td>com</td>
<td>D</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>Ifdux</td>
<td>Load Floating-Point Double with Update Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>631</td>
</tr>
<tr>
<td>Ifdx</td>
<td>Load Floating-Point Double Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>599</td>
</tr>
<tr>
<td>Ifq</td>
<td>Load Floating-Point Quad</td>
<td>POWER2</td>
<td>D</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>Ifqu</td>
<td>Load Floating-Point Quad with Update</td>
<td>POWER2</td>
<td>D</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>Ifqux</td>
<td>Load Floating-Point Quad with Update Indexed</td>
<td>POWER2</td>
<td>X</td>
<td>31</td>
<td>823</td>
</tr>
<tr>
<td>Ifqx</td>
<td>Load Floating-Point Quad Indexed</td>
<td>POWER2</td>
<td>X</td>
<td>31</td>
<td>791</td>
</tr>
<tr>
<td>Ifs</td>
<td>Load Floating-Point Single</td>
<td>com</td>
<td>D</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>Ifsu</td>
<td>Load Floating-Point Single with Update</td>
<td>com</td>
<td>D</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>Ifsux</td>
<td>Load Floating-Point Single with Update Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>567</td>
</tr>
<tr>
<td>Ifsx</td>
<td>Load Floating-Point Single Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>535</td>
</tr>
<tr>
<td>Iha</td>
<td>Load Half Algebraic</td>
<td>com</td>
<td>D</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>Ihau</td>
<td>Load Half Algebraic with Update</td>
<td>com</td>
<td>D</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>Ihaux</td>
<td>Load Half Algebraic with Update Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>375</td>
</tr>
<tr>
<td>Ihax</td>
<td>Load Half Algebraic Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>343</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Description</td>
<td>Mode</td>
<td>Index</td>
<td>Opcode</td>
<td>Size</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------------------------------</td>
<td>------</td>
<td>-------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>lhbrx</td>
<td>Load Half Byte-Reversed Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>790</td>
</tr>
<tr>
<td>lhz</td>
<td>Load Half and Zero</td>
<td>com</td>
<td>D</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>lhzu</td>
<td>Load Half and Zero with Update</td>
<td>com</td>
<td>D</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>lhzux</td>
<td>Load Half and Zero with Update Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>331</td>
</tr>
<tr>
<td>lhzx</td>
<td>Load Half and Zero Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>279</td>
</tr>
<tr>
<td>lm</td>
<td>Load Multiple</td>
<td>POWER family</td>
<td>D</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>lmw</td>
<td>Load Multiple Word</td>
<td>PowerPC</td>
<td>D</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>lscbx</td>
<td>Load String and Compare Byte Indexed</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>277</td>
</tr>
<tr>
<td>lsi</td>
<td>Load String Immediate</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>597</td>
</tr>
<tr>
<td>lswi</td>
<td>Load String Word Immediate</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>597</td>
</tr>
<tr>
<td>lswx</td>
<td>Load String Word Indexed</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>533</td>
</tr>
<tr>
<td>lsx</td>
<td>Load String Indexed</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>533</td>
</tr>
<tr>
<td>lu</td>
<td>Load with Update</td>
<td>POWER family</td>
<td>D</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>lux</td>
<td>Load with Update Indexed</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>55</td>
</tr>
<tr>
<td>lwax</td>
<td>Load Word and Reserve Indexed</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>20</td>
</tr>
<tr>
<td>lwbrx</td>
<td>Load Word Byte-Reversed Indexed</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>534</td>
</tr>
<tr>
<td>lwz</td>
<td>Load Word and Zero</td>
<td>PowerPC</td>
<td>D</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>lwzu</td>
<td>Load Word with Zero Update</td>
<td>PowerPC</td>
<td>D</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>lwzux</td>
<td>Load Word and Zero with Update Indexed</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>55</td>
</tr>
<tr>
<td>lwzx</td>
<td>Load Word and Zero Indexed</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>23</td>
</tr>
<tr>
<td>lx</td>
<td>Load Indexed</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>23</td>
</tr>
<tr>
<td>maskg[,]</td>
<td>Mask Generate</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>29</td>
</tr>
<tr>
<td>maskir[,]</td>
<td>Mask Insert from Register</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>541</td>
</tr>
<tr>
<td>mcrf</td>
<td>Move Condition Register Field</td>
<td>com</td>
<td>XL</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
<td>Architecture</td>
<td>Register</td>
<td>Type</td>
<td>Offset</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------</td>
<td>--------------</td>
<td>----------</td>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td>mcrfs</td>
<td>Move to Condition Register from FPSCR</td>
<td>com</td>
<td>X</td>
<td>63</td>
<td>64</td>
</tr>
<tr>
<td>mcrxr</td>
<td>Move to Condition Register from XER</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>512</td>
</tr>
<tr>
<td>mfcr</td>
<td>Move from Condition Register</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>19</td>
</tr>
<tr>
<td>mfcs[.]</td>
<td>Move from FPSCR</td>
<td>com</td>
<td>X</td>
<td>63</td>
<td>583</td>
</tr>
<tr>
<td>mfmsr</td>
<td>Move from Machine State Register</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>83</td>
</tr>
<tr>
<td>mfspr</td>
<td>Move from Special-Purpose Register</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>339</td>
</tr>
<tr>
<td>mfsr</td>
<td>Move from Segment Register</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>595</td>
</tr>
<tr>
<td>mfsri</td>
<td>Move from Segment Register Indirect</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>627</td>
</tr>
<tr>
<td>mfsrin</td>
<td>Move from Segment Register Indirect</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>659</td>
</tr>
<tr>
<td>mtcfr</td>
<td>Move to Condition Register Fields</td>
<td>com</td>
<td>XFX</td>
<td>31</td>
<td>144</td>
</tr>
<tr>
<td>mtfsb0[.]</td>
<td>Move to FPSCR Bit 0</td>
<td>com</td>
<td>X</td>
<td>63</td>
<td>70</td>
</tr>
<tr>
<td>mtfsb1[.]</td>
<td>Move to FPSCR Bit 1</td>
<td>com</td>
<td>X</td>
<td>63</td>
<td>38</td>
</tr>
<tr>
<td>mtfsf[.]</td>
<td>Move to FPSCR Fields</td>
<td>com</td>
<td>XFL</td>
<td>63</td>
<td>711</td>
</tr>
<tr>
<td>mtfsfi[.]</td>
<td>Move to FPSCR Field Immediate</td>
<td>com</td>
<td>X</td>
<td>63</td>
<td>134</td>
</tr>
<tr>
<td>mtmsr</td>
<td>Move to Machine State Register</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>146</td>
</tr>
<tr>
<td>mtspri</td>
<td>Move to Special-Purpose Register</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>467</td>
</tr>
<tr>
<td>mtssr</td>
<td>Move to Segment Register</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>210</td>
</tr>
<tr>
<td>mtssi</td>
<td>Move to Segment Register Indirect</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>242</td>
</tr>
<tr>
<td>mtssrin</td>
<td>Move to Segment Register Indirect</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>242</td>
</tr>
<tr>
<td>mul[o][.]</td>
<td>Multiply</td>
<td>POWER family</td>
<td>XO</td>
<td>31</td>
<td>107</td>
</tr>
<tr>
<td>mulhw[.]</td>
<td>Multiply High Word</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>75</td>
</tr>
<tr>
<td>mulhwu[.]</td>
<td>Multiply High Word Unsigned</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>11</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Description</td>
<td>FAMILY</td>
<td>OPERAND</td>
<td>CODE</td>
<td>PAGE</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------</td>
<td>--------------</td>
<td>---------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>muli</td>
<td>Multiply Immediate</td>
<td>POWER family</td>
<td>D</td>
<td>07</td>
<td></td>
</tr>
<tr>
<td>mulli</td>
<td>Multiply Low Immediate</td>
<td>PowerPC</td>
<td>D</td>
<td>07</td>
<td></td>
</tr>
<tr>
<td>mulw[0][]</td>
<td>Multiply Low Word</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>235</td>
</tr>
<tr>
<td>muls[0][]</td>
<td>Multiply Short</td>
<td>POWER family</td>
<td>XO</td>
<td>31</td>
<td>235</td>
</tr>
<tr>
<td>nabs[0][]</td>
<td>Negative Absolute</td>
<td>POWER family</td>
<td>XO</td>
<td>31</td>
<td>488</td>
</tr>
<tr>
<td>nand[]</td>
<td>NAND</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>476</td>
</tr>
<tr>
<td>neg[0][]</td>
<td>Negate</td>
<td>com</td>
<td>XO</td>
<td>31</td>
<td>104</td>
</tr>
<tr>
<td>nor[]</td>
<td>NOR</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>124</td>
</tr>
<tr>
<td>or[]</td>
<td>OR</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>444</td>
</tr>
<tr>
<td>orc[]</td>
<td>OR with Complement</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>412</td>
</tr>
<tr>
<td>ori</td>
<td>OR Immediate</td>
<td>PowerPC</td>
<td>D</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>oril</td>
<td>OR Immediate Lower</td>
<td>POWER family</td>
<td>D</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>oris</td>
<td>OR Immediate Shifted</td>
<td>PowerPC</td>
<td>D</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>oriu</td>
<td>OR Immediate Upper</td>
<td>POWER family</td>
<td>D</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>rac[]</td>
<td>Real Address Compute</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>818</td>
</tr>
<tr>
<td>rfi</td>
<td>Return from Interrupt</td>
<td>com</td>
<td>X</td>
<td>19</td>
<td>50</td>
</tr>
<tr>
<td>rfsvc</td>
<td>Return from SVC</td>
<td>POWER family</td>
<td>X</td>
<td>19</td>
<td>82</td>
</tr>
<tr>
<td>rlimi[]</td>
<td>Rotate Left Immediate then Mask Insert</td>
<td>POWER family</td>
<td>M</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>rlirmi[]</td>
<td>Rotate Left Immediate then AND with Mask</td>
<td>POWER family</td>
<td>M</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>rlimi[]</td>
<td>Rotate Left then Mask Insert</td>
<td>POWER family</td>
<td>M</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>rlnm[]</td>
<td>Rotate Left then AND with Mask</td>
<td>POWER family</td>
<td>M</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>rlwimi[]</td>
<td>Rotate Left Word Immediate then Mask Insert</td>
<td>PowerPC</td>
<td>M</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>rlwinm[]</td>
<td>Rotate Left Word Immediate then Mask Insert</td>
<td>PowerPC</td>
<td>M</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>rlwnm[]</td>
<td>Rotate Left Word then AND with Mask</td>
<td>PowerPC</td>
<td>M</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>rrib[]</td>
<td>Rotate Right and Insert Bit</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>537</td>
</tr>
<tr>
<td>sc</td>
<td>System Call</td>
<td>PowerPC</td>
<td>SC</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Family</td>
<td>Mode</td>
<td>Opcode</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>--------</td>
<td>------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td><code>sf[o]</code></td>
<td>Subtract from POWER family</td>
<td>XO</td>
<td>31</td>
<td>08</td>
<td></td>
</tr>
<tr>
<td><code>sf[e][o]</code></td>
<td>Subtract from Extended POWER family</td>
<td>XO</td>
<td>31</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td><code>sfi</code></td>
<td>Subtract from Immediate POWER family</td>
<td>D</td>
<td>08</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sfme[o][o]</code></td>
<td>Subtract from Minus One Extended POWER family</td>
<td>XO</td>
<td>31</td>
<td>232</td>
<td></td>
</tr>
<tr>
<td><code>sfze[o][o]</code></td>
<td>Subtract from Zero Extended POWER family</td>
<td>XO</td>
<td>31</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td><code>si</code></td>
<td>Subtract Immediate</td>
<td>com</td>
<td>D</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td><code>si.</code></td>
<td>Subtract Immediate and Record</td>
<td>com</td>
<td>D</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td><code>sl[.]</code></td>
<td>Shift Left POWER family</td>
<td>X</td>
<td>31</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td><code>sle[.]</code></td>
<td>Shift Left Extended POWER family</td>
<td>X</td>
<td>31</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td><code>sleq[.]</code></td>
<td>Shift Left Extended with MQ POWER family</td>
<td>X</td>
<td>31</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td><code>sliq[.]</code></td>
<td>Shift Left Immediate with MQ POWER family</td>
<td>X</td>
<td>31</td>
<td>184</td>
<td></td>
</tr>
<tr>
<td><code>slliq[.]</code></td>
<td>Shift Left Long Immediate with MQ POWER family</td>
<td>X</td>
<td>31</td>
<td>248</td>
<td></td>
</tr>
<tr>
<td><code>slq[.]</code></td>
<td>Shift Left Long with MQ POWER family</td>
<td>X</td>
<td>31</td>
<td>216</td>
<td></td>
</tr>
<tr>
<td><code>slq[.]</code></td>
<td>Shift Left with MQ POWER family</td>
<td>X</td>
<td>31</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td><code>slw[.]</code></td>
<td>Shift Left Word PowerPC</td>
<td>X</td>
<td>31</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td><code>sr[.]</code></td>
<td>Shift Right POWER family</td>
<td>X</td>
<td>31</td>
<td>536</td>
<td></td>
</tr>
<tr>
<td><code>sra[.]</code></td>
<td>Shift Right Algebraic POWER family</td>
<td>X</td>
<td>31</td>
<td>792</td>
<td></td>
</tr>
<tr>
<td><code>srai[.]</code></td>
<td>Shift Right Algebraic Immediate POWER family</td>
<td>X</td>
<td>31</td>
<td>824</td>
<td></td>
</tr>
<tr>
<td><code>sraiq[.]</code></td>
<td>Shift Right Algebraic, Immediate with MQ POWER family</td>
<td>X</td>
<td>31</td>
<td>952</td>
<td></td>
</tr>
<tr>
<td><code>sraq[.]</code></td>
<td>Shift Right Algebraic with MQ POWER family</td>
<td>X</td>
<td>31</td>
<td>920</td>
<td></td>
</tr>
<tr>
<td><code>sraw[.]</code></td>
<td>Shift Right Algebraic Word PowerPC</td>
<td>X</td>
<td>31</td>
<td>792</td>
<td></td>
</tr>
<tr>
<td><code>srawi[.]</code></td>
<td>Shift Right Algebraic Word Immediate PowerPC</td>
<td>X</td>
<td>31</td>
<td>824</td>
<td></td>
</tr>
<tr>
<td><code>sre[.]</code></td>
<td>Shift Right Extended POWER family</td>
<td>X</td>
<td>31</td>
<td>665</td>
<td></td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Description</td>
<td>Family</td>
<td>Mode</td>
<td>Size</td>
<td>Addressing</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>--------</td>
<td>------</td>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>srea[]</td>
<td>Shift Right Extended Algebraic</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>921</td>
</tr>
<tr>
<td>sreq[]</td>
<td>Shift Right Extended with MQ</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>729</td>
</tr>
<tr>
<td>sriq[]</td>
<td>Shift Right Immediate with MQ</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>696</td>
</tr>
<tr>
<td>srlq[]</td>
<td>Shift Right Long Immediate with MQ</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>760</td>
</tr>
<tr>
<td>srlq[]</td>
<td>Shift Right Long with MQ</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>728</td>
</tr>
<tr>
<td>srq[]</td>
<td>Shift Right with MQ</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>664</td>
</tr>
<tr>
<td>srw[]</td>
<td>Shift Right Word</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>536</td>
</tr>
<tr>
<td>st</td>
<td>Store</td>
<td>POWER family</td>
<td>D</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>stb</td>
<td>Store Byte</td>
<td>com</td>
<td>D</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>stbrx</td>
<td>Store Byte-Reversed Indexed</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>662</td>
</tr>
<tr>
<td>stbu</td>
<td>Store Byte with Update</td>
<td>com</td>
<td>D</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>stbux</td>
<td>Store Byte with Update Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>247</td>
</tr>
<tr>
<td>stbx</td>
<td>Store Byte Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>215</td>
</tr>
<tr>
<td>stfd</td>
<td>Store Floating-Point Double</td>
<td>com</td>
<td>D</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>stfdu</td>
<td>Store Floating-Point Double with Update</td>
<td>com</td>
<td>D</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>stfdux</td>
<td>Store Floating-Point Double with Update Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>759</td>
</tr>
<tr>
<td>stfdx</td>
<td>Store Floating-Point Double Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>727</td>
</tr>
<tr>
<td>stfiwx</td>
<td>Store Floating-Point as Integer Word Indexed</td>
<td>PPC opt.</td>
<td>X</td>
<td>31</td>
<td>983</td>
</tr>
<tr>
<td>stfq</td>
<td>Store Floating-Point Quad</td>
<td>POWER2</td>
<td>DS</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>stfqu</td>
<td>Store Floating-Point Quad with Update</td>
<td>POWER2</td>
<td>DS</td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Family</td>
<td>Mode</td>
<td>Index</td>
<td>Offset</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------------------------</td>
<td>-----------</td>
<td>------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>stfqux</td>
<td>Store Floating-Point Quad with Update Indexed</td>
<td>POWER2</td>
<td>X</td>
<td>31</td>
<td>951</td>
</tr>
<tr>
<td>stfqx</td>
<td>Store Floating-Point Quad Indexed</td>
<td>POWER2</td>
<td>X</td>
<td>31</td>
<td>919</td>
</tr>
<tr>
<td>stfs</td>
<td>Store Floating-Point Single</td>
<td>com</td>
<td>D</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>stfsu</td>
<td>Store Floating-Point Single with Update</td>
<td>com</td>
<td>D</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>stfsux</td>
<td>Store Floating-Point Single with Update Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>695</td>
</tr>
<tr>
<td>stfsx</td>
<td>Store Floating-Point Single Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>663</td>
</tr>
<tr>
<td>sth</td>
<td>Store Half</td>
<td>com</td>
<td>D</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>sthbrx</td>
<td>Store Half Byte-Reverse Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>918</td>
</tr>
<tr>
<td>sthu</td>
<td>Store Half with Update</td>
<td>com</td>
<td>D</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>sthux</td>
<td>Store Half with Update Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>439</td>
</tr>
<tr>
<td>sthx</td>
<td>Store Half Indexed</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>407</td>
</tr>
<tr>
<td>stm</td>
<td>Store Multiple</td>
<td>POWER family</td>
<td>D</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>strmw</td>
<td>Store Multiple Word</td>
<td>PowerPC</td>
<td>D</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>stsi</td>
<td>Store String Immediate</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>725</td>
</tr>
<tr>
<td>stswi</td>
<td>Store String Word Immediate</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>725</td>
</tr>
<tr>
<td>stswx</td>
<td>Store String Word Indexed</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>661</td>
</tr>
<tr>
<td>stsx</td>
<td>Store String Indexed</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>661</td>
</tr>
<tr>
<td>stu</td>
<td>Store with Update</td>
<td>POWER family</td>
<td>D</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>stux</td>
<td>Store with Update Indexed</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>183</td>
</tr>
<tr>
<td>stw</td>
<td>Store</td>
<td>PowerPC</td>
<td>D</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>stwbrx</td>
<td>Store Word Byte-Reverse Indexed</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>662</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Architecture</td>
<td>Mod</td>
<td>Offset</td>
<td>Opcode</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------------------------</td>
<td>--------------</td>
<td>-----</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>stwcx.</td>
<td>Store Word Conditional Indexed</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>150</td>
</tr>
<tr>
<td>stwu</td>
<td>Store Word with Update</td>
<td>PowerPC</td>
<td>D</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>stwux</td>
<td>Store Word with Update Indexed</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>183</td>
</tr>
<tr>
<td>stwx</td>
<td>Store Word Indexed</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>151</td>
</tr>
<tr>
<td>stx</td>
<td>Store Indexed</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>151</td>
</tr>
<tr>
<td>subf[0].[]</td>
<td>Subtract from</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>40</td>
</tr>
<tr>
<td>subfc[0].[]</td>
<td>Subtract from Carrying</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>08</td>
</tr>
<tr>
<td>subfe[0].[]</td>
<td>Subtract from Extended</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>136</td>
</tr>
<tr>
<td>subfic</td>
<td>Subtract from Immediate Carrying</td>
<td>PowerPC</td>
<td>D</td>
<td>08</td>
<td></td>
</tr>
<tr>
<td>subfme[0].[]</td>
<td>Subtract from Minus One Extended</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>232</td>
</tr>
<tr>
<td>subfze[0].[]</td>
<td>Subtract from Zero Extended</td>
<td>PowerPC</td>
<td>XO</td>
<td>31</td>
<td>200</td>
</tr>
<tr>
<td>svc[]][a]</td>
<td>Supervisor Call</td>
<td>POWER family</td>
<td>SC</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>sync</td>
<td>Synchronize</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>598</td>
</tr>
<tr>
<td>t</td>
<td>Trap</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>04</td>
</tr>
<tr>
<td>ti</td>
<td>Trap Immediate</td>
<td>POWER family</td>
<td>D</td>
<td>03</td>
<td></td>
</tr>
<tr>
<td>tlbi</td>
<td>Translation Look-aside Buffer Invalidate Entry</td>
<td>POWER family</td>
<td>X</td>
<td>31</td>
<td>306</td>
</tr>
<tr>
<td>tlbie</td>
<td>Translation Look-aside Buffer Invalidate Entry</td>
<td>PPC opt.</td>
<td>X</td>
<td>31</td>
<td>306</td>
</tr>
<tr>
<td>tblld</td>
<td>Load Data TLB Entry</td>
<td>603 only</td>
<td>X</td>
<td>31</td>
<td>978</td>
</tr>
<tr>
<td>tblli</td>
<td>Load Instruction TLB Entry</td>
<td>603 only</td>
<td>X</td>
<td>31</td>
<td>1010</td>
</tr>
<tr>
<td>tlbsync</td>
<td>Translation Look-aside Buffer Synchronize</td>
<td>PPC opt.</td>
<td>X</td>
<td>31</td>
<td>566</td>
</tr>
<tr>
<td>tw</td>
<td>Trap Word</td>
<td>PowerPC</td>
<td>X</td>
<td>31</td>
<td>04</td>
</tr>
<tr>
<td>twi</td>
<td>Trap Word Immediate</td>
<td>PowerPC</td>
<td>D</td>
<td>03</td>
<td></td>
</tr>
<tr>
<td>xor[.]</td>
<td>XOR</td>
<td>com</td>
<td>X</td>
<td>31</td>
<td>316</td>
</tr>
<tr>
<td>xori</td>
<td>XOR Immediate</td>
<td>PowerPC</td>
<td>D</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>xoril</td>
<td>XOR Immediate Lower</td>
<td>POWER family</td>
<td>D</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>xoris</td>
<td>XOR Immediate Shift</td>
<td>PowerPC</td>
<td>D</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>XOR Immediate Upper</td>
<td>POWER family</td>
<td>D</td>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------</td>
<td>---</td>
<td>----</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix C. Instruction Set Sorted by Primary and Extended Op Code

The Instruction Set Sorted by Primary and Extended Op Code table lists the instruction set, sorted first by primary op code and then by extended op code. The table column Implementation contains the following information:

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com</td>
<td>Supported by POWER family, POWER2, and PowerPC implementations.</td>
</tr>
<tr>
<td>POWER family</td>
<td>Supported only by POWER family and POWER2 implementations.</td>
</tr>
<tr>
<td>POWER2</td>
<td>Supported only by POWER2 implementations.</td>
</tr>
<tr>
<td>PowerPC</td>
<td>Supported only by PowerPC architecture.</td>
</tr>
<tr>
<td>PPC opt.</td>
<td>Defined only in PowerPC architecture and is an optional instruction.</td>
</tr>
<tr>
<td>603 only</td>
<td>Supported only on the PowerPC 603 RISC Microprocessor.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Set Sorted by Primary and Extended Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mnemonic</strong></td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>ti</td>
</tr>
<tr>
<td>twi</td>
</tr>
<tr>
<td>mulli</td>
</tr>
<tr>
<td>mulli</td>
</tr>
<tr>
<td>sfi</td>
</tr>
<tr>
<td>sfi.</td>
</tr>
<tr>
<td>dozi</td>
</tr>
<tr>
<td>cmpi</td>
</tr>
<tr>
<td>cmpli</td>
</tr>
<tr>
<td>addic</td>
</tr>
<tr>
<td>ai</td>
</tr>
<tr>
<td>si</td>
</tr>
<tr>
<td>addic.</td>
</tr>
<tr>
<td>si.</td>
</tr>
<tr>
<td>ai.</td>
</tr>
<tr>
<td>addi</td>
</tr>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>cal</td>
</tr>
<tr>
<td>addis</td>
</tr>
<tr>
<td>cau</td>
</tr>
<tr>
<td>bc[l][a]</td>
</tr>
<tr>
<td>sc</td>
</tr>
<tr>
<td>svc[l][a]</td>
</tr>
<tr>
<td>b[l][a]</td>
</tr>
<tr>
<td>mcrf</td>
</tr>
<tr>
<td>bclr[l]</td>
</tr>
<tr>
<td>bcr[l]</td>
</tr>
<tr>
<td>crnor</td>
</tr>
<tr>
<td>rfi</td>
</tr>
<tr>
<td>rfsvc</td>
</tr>
<tr>
<td>crandc</td>
</tr>
<tr>
<td>ics</td>
</tr>
<tr>
<td>isync</td>
</tr>
<tr>
<td>crxor</td>
</tr>
<tr>
<td>crnand</td>
</tr>
<tr>
<td>crand</td>
</tr>
<tr>
<td>creqv</td>
</tr>
<tr>
<td>crorc</td>
</tr>
<tr>
<td>cror</td>
</tr>
<tr>
<td>bcc[l]</td>
</tr>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>bcctr[]</td>
</tr>
<tr>
<td>rlimi[]</td>
</tr>
<tr>
<td>rlwimi[]</td>
</tr>
<tr>
<td>rlwinm[]</td>
</tr>
<tr>
<td>rlwinm[]</td>
</tr>
<tr>
<td>ori</td>
</tr>
<tr>
<td>oril</td>
</tr>
<tr>
<td>oris</td>
</tr>
<tr>
<td>oriu</td>
</tr>
<tr>
<td>xoril</td>
</tr>
<tr>
<td>xoris</td>
</tr>
<tr>
<td>xoriu</td>
</tr>
<tr>
<td>andi.</td>
</tr>
<tr>
<td>andi.</td>
</tr>
<tr>
<td>andis.</td>
</tr>
<tr>
<td>andiu.</td>
</tr>
<tr>
<td>cmp</td>
</tr>
<tr>
<td>t</td>
</tr>
<tr>
<td>tw</td>
</tr>
<tr>
<td>sf[o][.]</td>
</tr>
<tr>
<td>subfc[o][.]</td>
</tr>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>a[0][.]</td>
</tr>
<tr>
<td>addc[0][.]</td>
</tr>
<tr>
<td>mulhwr[.]</td>
</tr>
<tr>
<td>mfcr</td>
</tr>
<tr>
<td>lwvx</td>
</tr>
<tr>
<td>lwzx</td>
</tr>
<tr>
<td>l[ ]</td>
</tr>
<tr>
<td>sl[ ]</td>
</tr>
<tr>
<td>slw[ ]</td>
</tr>
<tr>
<td>cntlz[ ]</td>
</tr>
<tr>
<td>cntlzw[ ]</td>
</tr>
<tr>
<td>and[ ]</td>
</tr>
<tr>
<td>maskg[ ]</td>
</tr>
<tr>
<td>cmp[ ]</td>
</tr>
<tr>
<td>subf[0][ ]</td>
</tr>
<tr>
<td>dcbs[ ]</td>
</tr>
<tr>
<td>lux[ ]</td>
</tr>
<tr>
<td>lwzux[ ]</td>
</tr>
<tr>
<td>andc[ ]</td>
</tr>
<tr>
<td>mulhw[ ]</td>
</tr>
<tr>
<td>mfmsr</td>
</tr>
<tr>
<td>dcbs[ ]</td>
</tr>
<tr>
<td>lbzx[ ]</td>
</tr>
<tr>
<td>neg[0][ ]</td>
</tr>
<tr>
<td>mul[0][ ]</td>
</tr>
<tr>
<td>clf</td>
</tr>
<tr>
<td>lbzux[ ]</td>
</tr>
<tr>
<td>nor[ ]</td>
</tr>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>sfe[O][.]</td>
</tr>
<tr>
<td>subfe[O][.]</td>
</tr>
<tr>
<td>adde[O][.]</td>
</tr>
<tr>
<td>ae[O][.]</td>
</tr>
<tr>
<td>mtcfr</td>
</tr>
<tr>
<td>mtrmsr</td>
</tr>
<tr>
<td>stwcx.</td>
</tr>
<tr>
<td>stwx</td>
</tr>
<tr>
<td>stx</td>
</tr>
<tr>
<td>slq[.]</td>
</tr>
<tr>
<td>sle[.]</td>
</tr>
<tr>
<td>stux</td>
</tr>
<tr>
<td>stwux</td>
</tr>
<tr>
<td>silq[.]</td>
</tr>
<tr>
<td>sfze[O][.]</td>
</tr>
<tr>
<td>subfze[O][.]</td>
</tr>
<tr>
<td>addze[O][.]</td>
</tr>
<tr>
<td>aze[O][.]</td>
</tr>
<tr>
<td>mtsr</td>
</tr>
<tr>
<td>stbu</td>
</tr>
<tr>
<td>stbx</td>
</tr>
<tr>
<td>silq[.]</td>
</tr>
<tr>
<td>sleq[.]</td>
</tr>
<tr>
<td>sfme[O][.]</td>
</tr>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>subfme[0][.]</td>
</tr>
<tr>
<td>addme[0][.]</td>
</tr>
<tr>
<td>ame[0][.]</td>
</tr>
<tr>
<td>mullw[0][.]</td>
</tr>
<tr>
<td>muls[0][.]</td>
</tr>
<tr>
<td>mtsr</td>
</tr>
<tr>
<td>mtsrin</td>
</tr>
<tr>
<td>dcbtst</td>
</tr>
<tr>
<td>stbux</td>
</tr>
<tr>
<td>sliiq[.]</td>
</tr>
<tr>
<td>doz[0][.]</td>
</tr>
<tr>
<td>add[0][.]</td>
</tr>
<tr>
<td>cax[0][.]</td>
</tr>
<tr>
<td>lscbx</td>
</tr>
<tr>
<td>dcbt</td>
</tr>
<tr>
<td>lhzx</td>
</tr>
<tr>
<td>equiv[.]</td>
</tr>
<tr>
<td>tibi</td>
</tr>
<tr>
<td>tibie</td>
</tr>
<tr>
<td>eciwx</td>
</tr>
<tr>
<td>xor[.]</td>
</tr>
<tr>
<td>div[0][.]</td>
</tr>
<tr>
<td>lhzux</td>
</tr>
<tr>
<td>mfspr</td>
</tr>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>lhax</td>
</tr>
<tr>
<td>abs[o][]</td>
</tr>
<tr>
<td>divs[o][]</td>
</tr>
<tr>
<td>lhaux</td>
</tr>
<tr>
<td>sthx</td>
</tr>
<tr>
<td>orc[.]</td>
</tr>
<tr>
<td>ecowx</td>
</tr>
<tr>
<td>sthux</td>
</tr>
<tr>
<td>or[.]</td>
</tr>
<tr>
<td>divwu[o][]</td>
</tr>
<tr>
<td>mtspr</td>
</tr>
<tr>
<td>dcbi</td>
</tr>
<tr>
<td>nand[.]</td>
</tr>
<tr>
<td>nabs[o][]</td>
</tr>
<tr>
<td>divw[o][]</td>
</tr>
<tr>
<td>cli</td>
</tr>
<tr>
<td>mcrxr</td>
</tr>
<tr>
<td>clcs</td>
</tr>
<tr>
<td>lswx</td>
</tr>
<tr>
<td>lsx</td>
</tr>
<tr>
<td>lbpx</td>
</tr>
<tr>
<td>lwbrx</td>
</tr>
<tr>
<td>lfsx</td>
</tr>
<tr>
<td>sr[.]</td>
</tr>
<tr>
<td>srw[.]</td>
</tr>
<tr>
<td>Code</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>rrib[]</td>
</tr>
<tr>
<td>maskir[]</td>
</tr>
<tr>
<td>tilsync</td>
</tr>
<tr>
<td>lfsux</td>
</tr>
<tr>
<td>mfsr</td>
</tr>
<tr>
<td>lsi</td>
</tr>
<tr>
<td>lswl</td>
</tr>
<tr>
<td>dcs</td>
</tr>
<tr>
<td>sync</td>
</tr>
<tr>
<td>lfdx</td>
</tr>
<tr>
<td>mfsri</td>
</tr>
<tr>
<td>dclst</td>
</tr>
<tr>
<td>lfdux</td>
</tr>
<tr>
<td>mfsrin</td>
</tr>
<tr>
<td>stswx</td>
</tr>
<tr>
<td>stsx</td>
</tr>
<tr>
<td>stbrx</td>
</tr>
<tr>
<td>stwbrx</td>
</tr>
<tr>
<td>stfsx</td>
</tr>
<tr>
<td>srq[]</td>
</tr>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>sre[.]</td>
</tr>
<tr>
<td>stfsux</td>
</tr>
<tr>
<td>sriq[.]</td>
</tr>
<tr>
<td>stsi</td>
</tr>
<tr>
<td>stswi</td>
</tr>
<tr>
<td>stfdx</td>
</tr>
<tr>
<td>sriq[.]</td>
</tr>
<tr>
<td>sreq[.]</td>
</tr>
<tr>
<td>stfdux</td>
</tr>
<tr>
<td>sriq[.]</td>
</tr>
<tr>
<td>lhbrx</td>
</tr>
<tr>
<td>lfqx</td>
</tr>
<tr>
<td>sra[.]</td>
</tr>
<tr>
<td>sraw[.]</td>
</tr>
<tr>
<td>rac[.]</td>
</tr>
<tr>
<td>lfqux</td>
</tr>
<tr>
<td>sra[.]</td>
</tr>
<tr>
<td>sraw[.]</td>
</tr>
<tr>
<td>eieio</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>sthbrx</td>
</tr>
<tr>
<td>stfqx</td>
</tr>
<tr>
<td>sraq[]</td>
</tr>
<tr>
<td>srea[]</td>
</tr>
<tr>
<td>exts[]</td>
</tr>
<tr>
<td>extsh[]</td>
</tr>
<tr>
<td>stfqux</td>
</tr>
<tr>
<td>sraiq[]</td>
</tr>
<tr>
<td>extsb[]</td>
</tr>
<tr>
<td>tlibd</td>
</tr>
<tr>
<td>icbi</td>
</tr>
<tr>
<td>stfiwx</td>
</tr>
<tr>
<td>tlibli</td>
</tr>
<tr>
<td>dcbz</td>
</tr>
<tr>
<td>dclz</td>
</tr>
<tr>
<td>l</td>
</tr>
<tr>
<td>lwz</td>
</tr>
<tr>
<td>lu</td>
</tr>
<tr>
<td>lwzu</td>
</tr>
<tr>
<td>lbz</td>
</tr>
<tr>
<td>lbzu</td>
</tr>
<tr>
<td>st</td>
</tr>
<tr>
<td>stw</td>
</tr>
<tr>
<td>stu</td>
</tr>
<tr>
<td>mnemonic</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>stwu</td>
</tr>
<tr>
<td>stb</td>
</tr>
<tr>
<td>lhz</td>
</tr>
<tr>
<td>lhzu</td>
</tr>
<tr>
<td>lha</td>
</tr>
<tr>
<td>lhau</td>
</tr>
<tr>
<td>sth</td>
</tr>
<tr>
<td>sthu</td>
</tr>
<tr>
<td>lm</td>
</tr>
<tr>
<td>lmw</td>
</tr>
<tr>
<td>stm</td>
</tr>
<tr>
<td>stmw</td>
</tr>
<tr>
<td>lfs</td>
</tr>
<tr>
<td>lfsu</td>
</tr>
<tr>
<td>lfd</td>
</tr>
<tr>
<td>lfdu</td>
</tr>
<tr>
<td>stfs</td>
</tr>
<tr>
<td>stfsu</td>
</tr>
<tr>
<td>stfd</td>
</tr>
<tr>
<td>stfdu</td>
</tr>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>lfq</td>
</tr>
<tr>
<td>lfqu</td>
</tr>
<tr>
<td>fdivs[]</td>
</tr>
<tr>
<td>fsubs[]</td>
</tr>
<tr>
<td>fadds[]</td>
</tr>
<tr>
<td>fres[]</td>
</tr>
<tr>
<td>fmuls[]</td>
</tr>
<tr>
<td>fmsubs[]</td>
</tr>
<tr>
<td>fmadds[]</td>
</tr>
<tr>
<td>fnmsubs[]</td>
</tr>
<tr>
<td>fnmadds[]</td>
</tr>
<tr>
<td>stfq</td>
</tr>
<tr>
<td>stfqu</td>
</tr>
<tr>
<td>fcmpu</td>
</tr>
<tr>
<td>frsp[]</td>
</tr>
<tr>
<td>fcir[]</td>
</tr>
<tr>
<td>fctiw[]</td>
</tr>
<tr>
<td>fcirz[]</td>
</tr>
<tr>
<td>fctiwz[]</td>
</tr>
<tr>
<td>fd[]</td>
</tr>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>fdiv</td>
</tr>
<tr>
<td>fs</td>
</tr>
<tr>
<td>fsb</td>
</tr>
<tr>
<td>fa</td>
</tr>
<tr>
<td>fadd</td>
</tr>
<tr>
<td>fsqrt</td>
</tr>
<tr>
<td>fsel</td>
</tr>
<tr>
<td>fm</td>
</tr>
<tr>
<td>fmul</td>
</tr>
<tr>
<td>fmsq</td>
</tr>
<tr>
<td>fmsub</td>
</tr>
<tr>
<td>fma</td>
</tr>
<tr>
<td>fmadd</td>
</tr>
<tr>
<td>fnms</td>
</tr>
<tr>
<td>fnmsub</td>
</tr>
<tr>
<td>fnma</td>
</tr>
<tr>
<td>fnmadd</td>
</tr>
<tr>
<td>fcmpo</td>
</tr>
<tr>
<td>mtfsb1</td>
</tr>
<tr>
<td>fneg</td>
</tr>
<tr>
<td>mcrfs</td>
</tr>
<tr>
<td>mtfsb0</td>
</tr>
<tr>
<td>fmr</td>
</tr>
<tr>
<td>mtfsl</td>
</tr>
<tr>
<td>fnabs</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>fabs[]</td>
</tr>
<tr>
<td>mfs[]</td>
</tr>
<tr>
<td>mtfs[]</td>
</tr>
</tbody>
</table>
### Appendix D. Instructions Common to POWER family, POWER2, and PowerPC

#### Instructions Common to POWER family, POWER2, and PowerPC

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Format</th>
<th>Primary Op Code</th>
<th>Extended Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>and[,]</td>
<td>AND</td>
<td>X</td>
<td>31</td>
<td>28</td>
</tr>
<tr>
<td>andc[,]</td>
<td>AND with Complement</td>
<td>X</td>
<td>31</td>
<td>60</td>
</tr>
<tr>
<td>b[][a]</td>
<td>Branch</td>
<td>I</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>bc[[]][a]</td>
<td>Branch Conditional</td>
<td>B</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>cmp</td>
<td>Compare</td>
<td>X</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>cmpi</td>
<td>Compare Immediate</td>
<td>D</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>cmpl</td>
<td>Compare Logical</td>
<td>X</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>cmpli</td>
<td>Compare Logical Immediate</td>
<td>D</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>crand</td>
<td>Condition Register AND</td>
<td>XL</td>
<td>19</td>
<td>257</td>
</tr>
<tr>
<td>crandc</td>
<td>Condition Register AND with Complement</td>
<td>XL</td>
<td>19</td>
<td>129</td>
</tr>
<tr>
<td>creqv</td>
<td>Condition Register Equivalent</td>
<td>XL</td>
<td>19</td>
<td>289</td>
</tr>
<tr>
<td>crnand</td>
<td>Condition Register NAND</td>
<td>XL</td>
<td>19</td>
<td>225</td>
</tr>
<tr>
<td>crnor</td>
<td>Condition Register NOR</td>
<td>XL</td>
<td>19</td>
<td>33</td>
</tr>
<tr>
<td>cror</td>
<td>Condition Register OR</td>
<td>XL</td>
<td>19</td>
<td>449</td>
</tr>
<tr>
<td>crorc</td>
<td>Condition Register OR with Complement</td>
<td>XL</td>
<td>19</td>
<td>417</td>
</tr>
<tr>
<td>crxor</td>
<td>Condition Register XOR</td>
<td>XL</td>
<td>19</td>
<td>193</td>
</tr>
<tr>
<td>ecixw</td>
<td>External Control in Word Indexed</td>
<td>X</td>
<td>31</td>
<td>310</td>
</tr>
<tr>
<td>ecoxw</td>
<td>External Control out Word Indexed</td>
<td>X</td>
<td>31</td>
<td>438</td>
</tr>
<tr>
<td>eqv[,]</td>
<td>Equivalent</td>
<td>X</td>
<td>31</td>
<td>284</td>
</tr>
<tr>
<td>fabs[,]</td>
<td>Floating Absolute Value</td>
<td>X</td>
<td>63</td>
<td>264</td>
</tr>
<tr>
<td>fcmpo</td>
<td>Floating Compare Ordered</td>
<td>X</td>
<td>63</td>
<td>32</td>
</tr>
<tr>
<td>fcmpu</td>
<td>Floating Compare Unordered</td>
<td>XL</td>
<td>63</td>
<td>0</td>
</tr>
<tr>
<td>fmr[,]</td>
<td>Floating Move Register</td>
<td>X</td>
<td>63</td>
<td>72</td>
</tr>
<tr>
<td>fnabs[,]</td>
<td>Floating Negative Absolute Value</td>
<td>X</td>
<td>63</td>
<td>136</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>fneg[,]</td>
<td>Floating Negate</td>
<td>X</td>
<td>63</td>
<td>40</td>
</tr>
<tr>
<td>frsp[,]</td>
<td>Floating Round to Single Precision</td>
<td>X</td>
<td>63</td>
<td>12</td>
</tr>
<tr>
<td>lbz</td>
<td>Load Byte and Zero</td>
<td>D</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>lbzu</td>
<td>Load Byte and Zero with Update</td>
<td>D</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>lbzux</td>
<td>Load Byte and Zero with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>119</td>
</tr>
<tr>
<td>lbzx</td>
<td>Load Byte and Zero Indexed</td>
<td>X</td>
<td>31</td>
<td>87</td>
</tr>
<tr>
<td>lfd</td>
<td>Load Floating-Point Double</td>
<td>D</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>lfdu</td>
<td>Load Floating-Point Double with Update</td>
<td>D</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>lfdux</td>
<td>Load Floating-Point Double with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>631</td>
</tr>
<tr>
<td>lfdx</td>
<td>Load Floating-Point Double Indexed</td>
<td>X</td>
<td>31</td>
<td>599</td>
</tr>
<tr>
<td>lfs</td>
<td>Load Floating-Point Single</td>
<td>D</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>lfsu</td>
<td>Load Floating-Point Single with Update</td>
<td>D</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>lfsux</td>
<td>Load Floating-Point Single with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>567</td>
</tr>
<tr>
<td>lfsx</td>
<td>Load Floating-Point Single Indexed</td>
<td>X</td>
<td>31</td>
<td>535</td>
</tr>
<tr>
<td>lha</td>
<td>Load Half Algebraic</td>
<td>D</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>lhau</td>
<td>Load Half Algebraic with Update</td>
<td>D</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>lhaux</td>
<td>Load Half Algebraic with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>375</td>
</tr>
<tr>
<td>lhax</td>
<td>Load Half Algebraic Indexed</td>
<td>X</td>
<td>31</td>
<td>343</td>
</tr>
<tr>
<td>lhbrx</td>
<td>Load Half Byte-Reversed Indexed</td>
<td>X</td>
<td>31</td>
<td>790</td>
</tr>
<tr>
<td>lhz</td>
<td>Load Half and Zero</td>
<td>D</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>lhzu</td>
<td>Load Half and Zero with Update</td>
<td>D</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>lhzux</td>
<td>Load Half and Zero with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>331</td>
</tr>
<tr>
<td>lhzx</td>
<td>Load Half and Zero Indexed</td>
<td>X</td>
<td>31</td>
<td>279</td>
</tr>
<tr>
<td>mcrf</td>
<td>Move Condition Register Field</td>
<td>XL</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>mcrfs</td>
<td>Move to Condition Register from FPSCR</td>
<td>X</td>
<td>63</td>
<td>64</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>R</td>
<td>R2</td>
<td>Opcode</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------</td>
<td>----</td>
<td>----</td>
<td>--------</td>
</tr>
<tr>
<td>mcrxr</td>
<td>Move to Condition Register from XER</td>
<td>X</td>
<td>31</td>
<td>512</td>
</tr>
<tr>
<td>mfcr</td>
<td>Move from Condition Register</td>
<td>X</td>
<td>31</td>
<td>19</td>
</tr>
<tr>
<td>mffs[]</td>
<td>Move from FPSCR</td>
<td>X</td>
<td>63</td>
<td>583</td>
</tr>
<tr>
<td>mfmsr</td>
<td>Move from Machine State Register</td>
<td>X</td>
<td>31</td>
<td>83</td>
</tr>
<tr>
<td>mfsp r</td>
<td>Move from Special-Purpose Register</td>
<td>X</td>
<td>31</td>
<td>339</td>
</tr>
<tr>
<td>mfsr</td>
<td>Move from Segment Register</td>
<td>X</td>
<td>31</td>
<td>595</td>
</tr>
<tr>
<td>mtcrf</td>
<td>Move to Condition Register Fields</td>
<td>XFX</td>
<td>31</td>
<td>144</td>
</tr>
<tr>
<td>mtfsb0[]</td>
<td>Move to FPSCR Bit 0</td>
<td>X</td>
<td>63</td>
<td>70</td>
</tr>
<tr>
<td>mtfsb1[]</td>
<td>Move to FPSCR Bit 1</td>
<td>X</td>
<td>63</td>
<td>38</td>
</tr>
<tr>
<td>mtfsf[]</td>
<td>Move to FPSCR Fields</td>
<td>XFL</td>
<td>63</td>
<td>711</td>
</tr>
<tr>
<td>mtfsfi[]</td>
<td>Move to FPSCR Field Immediate</td>
<td>X</td>
<td>63</td>
<td>134</td>
</tr>
<tr>
<td>mtmsr</td>
<td>Move to Machine State Register</td>
<td>X</td>
<td>31</td>
<td>146</td>
</tr>
<tr>
<td>mtspr</td>
<td>Move to Special-Purpose Register</td>
<td>X</td>
<td>31</td>
<td>467</td>
</tr>
<tr>
<td>mtsr</td>
<td>Move to Segment Register</td>
<td>X</td>
<td>31</td>
<td>210</td>
</tr>
<tr>
<td>nand[]</td>
<td>NAND</td>
<td>X</td>
<td>31</td>
<td>476</td>
</tr>
<tr>
<td>nego[]</td>
<td>Negate</td>
<td>XO</td>
<td>31</td>
<td>104</td>
</tr>
<tr>
<td>nor[]</td>
<td>NOR</td>
<td>X</td>
<td>31</td>
<td>124</td>
</tr>
<tr>
<td>or[]</td>
<td>OR</td>
<td>X</td>
<td>31</td>
<td>444</td>
</tr>
<tr>
<td>orc[]</td>
<td>OR with Complement</td>
<td>X</td>
<td>31</td>
<td>412</td>
</tr>
<tr>
<td>rfi</td>
<td>Return from Interrupt</td>
<td>X</td>
<td>19</td>
<td>50</td>
</tr>
<tr>
<td>si</td>
<td>Subtract Immediate</td>
<td>D</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>si.</td>
<td>Subtract Immediate and Record</td>
<td>D</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>stb</td>
<td>Store Byte</td>
<td>D</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>stbu</td>
<td>Store Byte with Update</td>
<td>D</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>stbux</td>
<td>Store Byte with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>247</td>
</tr>
<tr>
<td>stbx</td>
<td>Store Byte Indexed</td>
<td>X</td>
<td>31</td>
<td>215</td>
</tr>
<tr>
<td>stfd</td>
<td>Store Floating-Point Double</td>
<td>D</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>stfd u</td>
<td>Store Floating-Point Double with Update</td>
<td>D</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Modifier</td>
<td>Size</td>
<td>Value</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------</td>
<td>----------</td>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>stfdux</td>
<td>Store Floating-Point Double with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>759</td>
</tr>
<tr>
<td>stfdx</td>
<td>Store Floating-Point Double Indexed</td>
<td>X</td>
<td>31</td>
<td>727</td>
</tr>
<tr>
<td>stfs</td>
<td>Store Floating-Point Single</td>
<td>D</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>stfsu</td>
<td>Store Floating-Point Single with Update</td>
<td>D</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>stfsux</td>
<td>Store Floating-Point Single with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>695</td>
</tr>
<tr>
<td>stfsx</td>
<td>Store Floating-Point Single Indexed</td>
<td>X</td>
<td>31</td>
<td>663</td>
</tr>
<tr>
<td>sth</td>
<td>Store Half</td>
<td>D</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>sthbrx</td>
<td>Store Half Byte-Reverse Indexed</td>
<td>X</td>
<td>31</td>
<td>918</td>
</tr>
<tr>
<td>sthu</td>
<td>Store Half with Update</td>
<td>D</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>sthux</td>
<td>Store Half with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>439</td>
</tr>
<tr>
<td>sthx</td>
<td>Store Half Indexed</td>
<td>X</td>
<td>31</td>
<td>407</td>
</tr>
<tr>
<td>xor[,]</td>
<td>XOR</td>
<td>X</td>
<td>31</td>
<td>316</td>
</tr>
</tbody>
</table>
Appendix E. POWER family and POWER2 Instructions

In the following POWER family and POWER2 Instructions table, Instructions that are supported only in POWER2 implementations are indicated by “POWER2” in the POWER2 Only column:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a[o][]</td>
<td>Add Carrying</td>
<td>XO</td>
<td>31</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>abs[o][]</td>
<td>Absolute</td>
<td>XO</td>
<td>31</td>
<td>360</td>
<td></td>
</tr>
<tr>
<td>ae[o][]</td>
<td>Add Extended</td>
<td>XO</td>
<td>31</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>ai</td>
<td>Add Immediate</td>
<td>D</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ai.</td>
<td>Add Immediate and Record</td>
<td>D</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ame[o][]</td>
<td>Add to Minus One Extended</td>
<td>XO</td>
<td>31</td>
<td>234</td>
<td></td>
</tr>
<tr>
<td>and[]</td>
<td>AND</td>
<td>X</td>
<td>31</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>andc[]</td>
<td>AND with Complement</td>
<td>X</td>
<td>31</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>andil.</td>
<td>AND Immediate Lower</td>
<td>D</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>andiu.</td>
<td>AND Immediate Upper</td>
<td>D</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aze[o][]</td>
<td>Add to Zero Extended</td>
<td>XO</td>
<td>31</td>
<td>202</td>
<td></td>
</tr>
<tr>
<td>b[l][a]</td>
<td>Branch</td>
<td>I</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bc[l][a]</td>
<td>Branch Conditional</td>
<td>B</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bcc[l]</td>
<td>Branch Conditional to Count Register</td>
<td>XL</td>
<td>19</td>
<td>528</td>
<td></td>
</tr>
<tr>
<td>bcr[l]</td>
<td>Branch Conditional Register</td>
<td>XL</td>
<td>19</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>cal</td>
<td>Compute Address Lower</td>
<td>D</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cau</td>
<td>Compute Address Upper</td>
<td>D</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cax[o][]</td>
<td>Compute Address</td>
<td>XO</td>
<td>31</td>
<td>266</td>
<td></td>
</tr>
<tr>
<td>clcs</td>
<td>Cache Line Compute Size</td>
<td>X</td>
<td>31</td>
<td>531</td>
<td></td>
</tr>
<tr>
<td>clf</td>
<td>Cache Line Flush</td>
<td>X</td>
<td>31</td>
<td>118</td>
<td></td>
</tr>
<tr>
<td>cli</td>
<td>Cache Line Invalidate</td>
<td>X</td>
<td>31</td>
<td>502</td>
<td></td>
</tr>
<tr>
<td>cmp</td>
<td>Compare</td>
<td>X</td>
<td>31</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>cmpi</td>
<td>Compare Immediate</td>
<td>D</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpl</td>
<td>Compare Logical</td>
<td>X</td>
<td>31</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
<td>Op Code</td>
<td>Size</td>
<td>Address Size</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------</td>
<td>---------</td>
<td>------</td>
<td>--------------</td>
<td></td>
</tr>
<tr>
<td>cmpli</td>
<td>Compare Logical Immediate</td>
<td>D</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cntlz[.]</td>
<td>Count Leading Zeros</td>
<td>X</td>
<td>31</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>crand</td>
<td>Condition Register AND</td>
<td>XL</td>
<td>19</td>
<td>257</td>
<td></td>
</tr>
<tr>
<td>crandc</td>
<td>Condition Register AND with Complement</td>
<td>XL</td>
<td>19</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>creqv</td>
<td>Condition Register Equivalent</td>
<td>XL</td>
<td>19</td>
<td>289</td>
<td></td>
</tr>
<tr>
<td>crnand</td>
<td>Condition Register NAND</td>
<td>XL</td>
<td>19</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>crnor</td>
<td>Condition Register NOR</td>
<td>XL</td>
<td>19</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>cror</td>
<td>Condition Register OR</td>
<td>XL</td>
<td>19</td>
<td>449</td>
<td></td>
</tr>
<tr>
<td>crorc</td>
<td>Condition Register OR with Complement</td>
<td>XL</td>
<td>19</td>
<td>417</td>
<td></td>
</tr>
<tr>
<td>crxor</td>
<td>Condition Register XOR</td>
<td>XL</td>
<td>19</td>
<td>193</td>
<td></td>
</tr>
<tr>
<td>dclst</td>
<td>Data Cache Line Store</td>
<td>X</td>
<td>31</td>
<td>630</td>
<td></td>
</tr>
<tr>
<td>dclz</td>
<td>Data Cache Line Set to Zero</td>
<td>X</td>
<td>31</td>
<td>1014</td>
<td></td>
</tr>
<tr>
<td>dcs</td>
<td>Data Cache Synchronize</td>
<td>X</td>
<td>31</td>
<td>598</td>
<td></td>
</tr>
<tr>
<td>div[0][.]</td>
<td>Divide</td>
<td>XO</td>
<td>31</td>
<td>331</td>
<td></td>
</tr>
<tr>
<td>divs[0][.</td>
<td></td>
<td>Divide Short</td>
<td>XO</td>
<td>31</td>
<td>363</td>
</tr>
<tr>
<td>doz[0][.]</td>
<td>Difference or Zero Immediate</td>
<td>XO</td>
<td>31</td>
<td>264</td>
<td></td>
</tr>
<tr>
<td>dozi</td>
<td>Difference or Zero Immediate</td>
<td>D</td>
<td>09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eciwx</td>
<td>External Control in Word Indexed</td>
<td>X</td>
<td>31</td>
<td>310</td>
<td></td>
</tr>
<tr>
<td>ecowx</td>
<td>External Control out Word Indexed</td>
<td>X</td>
<td>31</td>
<td>438</td>
<td></td>
</tr>
<tr>
<td>eqv[.]</td>
<td>Equivalent</td>
<td>X</td>
<td>31</td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>exts[.]</td>
<td>Extend Sign</td>
<td>X</td>
<td>31</td>
<td>922</td>
<td></td>
</tr>
<tr>
<td>fa[.]</td>
<td>Floating Add</td>
<td>A</td>
<td>63</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>fabs[.]</td>
<td>Floating Absolute Value</td>
<td>X</td>
<td>63</td>
<td>264</td>
<td></td>
</tr>
<tr>
<td>fcir[.]</td>
<td>Floating Convert to Integer Word</td>
<td>X</td>
<td>63</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>fcirz[.]</td>
<td>Floating Convert to Integer Word with Round to Zero</td>
<td>X</td>
<td>63</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Class</td>
<td>Opcode</td>
<td>Size</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------</td>
<td>-------</td>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>fcmpo</td>
<td>Floating Compare</td>
<td>X</td>
<td>63</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>fcmpu</td>
<td>Floating Compare Unordered</td>
<td>XL</td>
<td>63</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>fd[]</td>
<td>Floating Divide</td>
<td>A</td>
<td>63</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>fm[]</td>
<td>Floating Multiply</td>
<td>A</td>
<td>63</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>fma[]</td>
<td>Floating Multiply-Add</td>
<td>A</td>
<td>63</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>fmrf[]</td>
<td>Floating Move Register</td>
<td>X</td>
<td>63</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>fms[]</td>
<td>Floating Multiply-Subtract</td>
<td>A</td>
<td>63</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>fnabs[]</td>
<td>Floating Negative Absolute Value</td>
<td>X</td>
<td>63</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>fneg[]</td>
<td>Floating Negate</td>
<td>X</td>
<td>63</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>fnma[]</td>
<td>Floating Negative Multiply-Add</td>
<td>A</td>
<td>63</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>fnms[]</td>
<td>Floating Negative Multiply-Subtract</td>
<td>A</td>
<td>63</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>frsp[]</td>
<td>Floating Round to Single Precision</td>
<td>X</td>
<td>63</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>fs[]</td>
<td>Floating Subtract</td>
<td>A</td>
<td>63</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>fsgt[]</td>
<td>Floating Square Root</td>
<td>POWER2</td>
<td>A</td>
<td>63</td>
<td>22</td>
</tr>
<tr>
<td>ics</td>
<td>Instruction Cache Synchronize</td>
<td>X</td>
<td>19</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>Load</td>
<td>D</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lbrx</td>
<td>Load Byte-Reversed Indexed</td>
<td>X</td>
<td>31</td>
<td>534</td>
<td></td>
</tr>
<tr>
<td>lbz</td>
<td>Load Byte and Zero</td>
<td>D</td>
<td>34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lbzu</td>
<td>Load Byte and Zero with Update</td>
<td>D</td>
<td>35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lbzux</td>
<td>Load Byte and Zero with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>119</td>
<td></td>
</tr>
<tr>
<td>lbzx</td>
<td>Load Byte and Zero Indexed</td>
<td>X</td>
<td>31</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>lfd</td>
<td>Load Floating-Point Double</td>
<td>D</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lfdu</td>
<td>Load Floating-Point Double with Update</td>
<td>D</td>
<td>51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lfdux</td>
<td>Load Floating-Point Double with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>631</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Mode</td>
<td>Offset</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------------------</td>
<td>------</td>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>ldx</td>
<td>Load Floating-Point Double Indexed</td>
<td>X</td>
<td>31</td>
<td>599</td>
<td></td>
</tr>
<tr>
<td>ifq</td>
<td>Load Floating-Point Quad</td>
<td>POWER2</td>
<td>D</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>ifqu</td>
<td>Load Floating-Point Quad with Update</td>
<td>POWER2</td>
<td>D</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>ifqux</td>
<td>Load Floating-Point Quad with Update Indexed</td>
<td>POWER2</td>
<td>X</td>
<td>31</td>
<td>823</td>
</tr>
<tr>
<td>ifqy</td>
<td>Load Floating-Point Quad Indexed</td>
<td>POWER2</td>
<td>X</td>
<td>31</td>
<td>791</td>
</tr>
<tr>
<td>ifs</td>
<td>Load Floating-Point Single</td>
<td>D</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ifsu</td>
<td>Load Floating-Point Single with Update</td>
<td>D</td>
<td>49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ifsux</td>
<td>Load Floating-Point Single with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>567</td>
<td></td>
</tr>
<tr>
<td>ifsx</td>
<td>Load Floating-Point Single Indexed</td>
<td>X</td>
<td>31</td>
<td>535</td>
<td></td>
</tr>
<tr>
<td>lha</td>
<td>Load Half Algebraic</td>
<td>D</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lhaux</td>
<td>Load Half Algebraic with Update</td>
<td>D</td>
<td>43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lhauxx</td>
<td>Load Half Algebraic with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>375</td>
<td></td>
</tr>
<tr>
<td>lhx</td>
<td>Load Half Algebraic</td>
<td>X</td>
<td>31</td>
<td>343</td>
<td></td>
</tr>
<tr>
<td>lhbx</td>
<td>Load Half Indexed</td>
<td>X</td>
<td>31</td>
<td>790</td>
<td></td>
</tr>
<tr>
<td>lhz</td>
<td>Load Half and Zero</td>
<td>D</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lhzux</td>
<td>Load Half and Zero with Update</td>
<td>D</td>
<td>41</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lhzuxx</td>
<td>Load Half and Zero with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>331</td>
<td></td>
</tr>
<tr>
<td>lhzx</td>
<td>Load Half and Zero Indexed</td>
<td>X</td>
<td>31</td>
<td>279</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Mode</td>
<td>Size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
<td>------</td>
<td>------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>lm</td>
<td>Load Multiple</td>
<td>D</td>
<td>46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lscbx</td>
<td>Load String and Compare Byte Indexed</td>
<td>X</td>
<td>31</td>
<td>277</td>
<td></td>
</tr>
<tr>
<td>lsi</td>
<td>Load String Immediate</td>
<td>X</td>
<td>31</td>
<td>597</td>
<td></td>
</tr>
<tr>
<td>lsx</td>
<td>Load String Indexed</td>
<td>X</td>
<td>31</td>
<td>533</td>
<td></td>
</tr>
<tr>
<td>lu</td>
<td>Load with Update</td>
<td>D</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lux</td>
<td>Load with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>lx</td>
<td>Load Indexed</td>
<td>X</td>
<td>31</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>maskg[.]</td>
<td>Mask Generate</td>
<td>X</td>
<td>31</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>maskir[.]</td>
<td>Mask Insert from Register</td>
<td>X</td>
<td>31</td>
<td>541</td>
<td></td>
</tr>
<tr>
<td>mcrf</td>
<td>Move Condition Register Field</td>
<td>XL</td>
<td>19</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>mcrfs</td>
<td>Move to Condition Register from FPSCR</td>
<td>X</td>
<td>63</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>mcrxr</td>
<td>Move to Condition Register from XER</td>
<td>X</td>
<td>31</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>mfcr</td>
<td>Move from Condition Register</td>
<td>X</td>
<td>31</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>mffs[.]</td>
<td>Move from FPSCR</td>
<td>X</td>
<td>63</td>
<td>583</td>
<td></td>
</tr>
<tr>
<td>mfmsr</td>
<td>Move from Machine State Register</td>
<td>X</td>
<td>31</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>mfspkr</td>
<td>Move from Special-Purpose Register</td>
<td>X</td>
<td>31</td>
<td>339</td>
<td></td>
</tr>
<tr>
<td>mfsr</td>
<td>Move from Segment Register</td>
<td>X</td>
<td>31</td>
<td>595</td>
<td></td>
</tr>
<tr>
<td>mfsri</td>
<td>Move from Segment Register Indirect</td>
<td>X</td>
<td>31</td>
<td>627</td>
<td></td>
</tr>
<tr>
<td>mtcrf</td>
<td>Move to Condition Register Fields</td>
<td>XFX</td>
<td>31</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>mtfsb0[.]</td>
<td>Move to FPSCR Bit 0</td>
<td>X</td>
<td>63</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>mtfsb1[.]</td>
<td>Move to FPSCR Bit 1</td>
<td>X</td>
<td>63</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>mtfsf[.]</td>
<td>Move to FPSCR Fields</td>
<td>XFL</td>
<td>63</td>
<td>711</td>
<td></td>
</tr>
<tr>
<td>mtfsfl[.]</td>
<td>Move to FPSCR Field Immediate</td>
<td>X</td>
<td>63</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
<td>Operand</td>
<td>Mode</td>
<td>Address</td>
<td>mnemonic</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------</td>
<td>---------</td>
<td>------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>mtmsr</td>
<td>Move to Machine State Register</td>
<td>X</td>
<td>31</td>
<td>146</td>
<td>mtmsr</td>
</tr>
<tr>
<td>mtspri</td>
<td>Move to Special-Purpose Register</td>
<td>X</td>
<td>31</td>
<td>467</td>
<td>mtspri</td>
</tr>
<tr>
<td>mtsr</td>
<td>Move to Segment Register</td>
<td>X</td>
<td>31</td>
<td>210</td>
<td>mtsr</td>
</tr>
<tr>
<td>mtsri</td>
<td>Move to Segment Register Indirect</td>
<td>X</td>
<td>31</td>
<td>242</td>
<td>mtsri</td>
</tr>
<tr>
<td>mul[O][.]</td>
<td>Multiply</td>
<td>XO</td>
<td>31</td>
<td>107</td>
<td>mul[O][.]</td>
</tr>
<tr>
<td>mull</td>
<td>Multiply Immediate</td>
<td>D</td>
<td>07</td>
<td></td>
<td>mull</td>
</tr>
<tr>
<td>muls[O][.</td>
<td></td>
<td>Multiply Short</td>
<td>XO</td>
<td>31</td>
<td>235</td>
</tr>
<tr>
<td>nabs[O][.</td>
<td></td>
<td>Negative Absolute</td>
<td>XO</td>
<td>31</td>
<td>488</td>
</tr>
<tr>
<td>nand[.]</td>
<td>NAND</td>
<td>X</td>
<td>31</td>
<td>476</td>
<td>nand[.]</td>
</tr>
<tr>
<td>neg[O][.</td>
<td></td>
<td>Negate</td>
<td>XO</td>
<td>31</td>
<td>104</td>
</tr>
<tr>
<td>nor[.]</td>
<td>NOR</td>
<td>X</td>
<td>31</td>
<td>124</td>
<td>nor[.]</td>
</tr>
<tr>
<td>or[.]</td>
<td>OR</td>
<td>X</td>
<td>31</td>
<td>444</td>
<td>or[.]</td>
</tr>
<tr>
<td>orc[.]</td>
<td>OR with Complement</td>
<td>X</td>
<td>31</td>
<td>412</td>
<td>orc[.]</td>
</tr>
<tr>
<td>oril</td>
<td>OR Immediate Lower</td>
<td>D</td>
<td>24</td>
<td></td>
<td>oril</td>
</tr>
<tr>
<td>oriu</td>
<td>OR Immediate Upper</td>
<td>D</td>
<td>25</td>
<td></td>
<td>oriu</td>
</tr>
<tr>
<td>rac[.]</td>
<td>Real Address Compute</td>
<td>X</td>
<td>31</td>
<td>818</td>
<td>rac[.]</td>
</tr>
<tr>
<td>rfi</td>
<td>Return from Interrupt</td>
<td>X</td>
<td>19</td>
<td>50</td>
<td>rfi</td>
</tr>
<tr>
<td>rfsvc</td>
<td>Return from SVC</td>
<td>X</td>
<td>19</td>
<td>82</td>
<td>rfsvc</td>
</tr>
<tr>
<td>rlim[.</td>
<td></td>
<td>Rotate Left Immediate then Mask Insert</td>
<td>M</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>rlinm[.</td>
<td></td>
<td>Rotate Left Immediate then AND with Mask</td>
<td>M</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>rlim[.</td>
<td></td>
<td>Rotate Left then Mask Insert</td>
<td>M</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>rlinm[.</td>
<td></td>
<td>Rotate Left then AND with Mask</td>
<td>M</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>rrib[.</td>
<td></td>
<td>Rotate Right and Insert Bit</td>
<td>X</td>
<td>31</td>
<td>537</td>
</tr>
<tr>
<td>sf[O][.</td>
<td></td>
<td>Subtract from</td>
<td>XO</td>
<td>31</td>
<td>08</td>
</tr>
<tr>
<td>sfe[O][.</td>
<td></td>
<td>Subtract from Extended</td>
<td>XO</td>
<td>31</td>
<td>136</td>
</tr>
<tr>
<td>sfi</td>
<td>Subtract from Immediate</td>
<td>D</td>
<td>08</td>
<td></td>
<td>sfi</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Rox</td>
<td>Op0</td>
<td>Op1</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>sfme[0][]</td>
<td>Subtract from Minus One Extended</td>
<td>XO</td>
<td>31</td>
<td>232</td>
<td></td>
</tr>
<tr>
<td>sfze[0][]</td>
<td>Subtract from Zero Extended</td>
<td>XO</td>
<td>31</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>si</td>
<td>Subtract Immediate</td>
<td>D</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>si[]</td>
<td>Subtract Immediate and Record</td>
<td>D</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>si[.]</td>
<td>Shift Left</td>
<td>X</td>
<td>31</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>sie[.]</td>
<td>Shift Left Extended</td>
<td>X</td>
<td>31</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>sleq[.]</td>
<td>Shift Left Extended with MQ</td>
<td>X</td>
<td>31</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>sliq[.]</td>
<td>Shift Left Immediate with MQ</td>
<td>X</td>
<td>31</td>
<td>184</td>
<td></td>
</tr>
<tr>
<td>sliq[]</td>
<td>Shift Left Long Immediate with MQ</td>
<td>X</td>
<td>31</td>
<td>248</td>
<td></td>
</tr>
<tr>
<td>silq[]</td>
<td>Shift Left Long with MQ</td>
<td>X</td>
<td>31</td>
<td>216</td>
<td></td>
</tr>
<tr>
<td>slq[.]</td>
<td>Shift Left with MQ</td>
<td>X</td>
<td>31</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td>sr[.]</td>
<td>Shift Right</td>
<td>X</td>
<td>31</td>
<td>536</td>
<td></td>
</tr>
<tr>
<td>sra[.]</td>
<td>Shift Right Algebraic</td>
<td>X</td>
<td>31</td>
<td>792</td>
<td></td>
</tr>
<tr>
<td>sral[.]</td>
<td>Shift Right Algebraic Immediate</td>
<td>X</td>
<td>31</td>
<td>824</td>
<td></td>
</tr>
<tr>
<td>sraiq[.]</td>
<td>Shift Right Algebraic Immediate with MQ</td>
<td>X</td>
<td>31</td>
<td>952</td>
<td></td>
</tr>
<tr>
<td>sraq[.]</td>
<td>Shift Right Algebraic with MQ</td>
<td>X</td>
<td>31</td>
<td>920</td>
<td></td>
</tr>
<tr>
<td>sre[.]</td>
<td>Shift Right Extended</td>
<td>X</td>
<td>31</td>
<td>665</td>
<td></td>
</tr>
<tr>
<td>srea[.]</td>
<td>Shift Right Extended Algebraic</td>
<td>X</td>
<td>31</td>
<td>921</td>
<td></td>
</tr>
<tr>
<td>sreq[.]</td>
<td>Shift Right Extended with MQ</td>
<td>X</td>
<td>31</td>
<td>729</td>
<td></td>
</tr>
<tr>
<td>sriq[.]</td>
<td>Shift Right Immediate with MQ</td>
<td>X</td>
<td>31</td>
<td>696</td>
<td></td>
</tr>
<tr>
<td>srliq[.]</td>
<td>Shift Right Long Immediate with MQ</td>
<td>X</td>
<td>31</td>
<td>760</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Mod</td>
<td>Size</td>
<td>Offset</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
<td>-----</td>
<td>------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>srlq[.]</td>
<td>Shift Right Long with MQ</td>
<td>X</td>
<td>31</td>
<td>728</td>
<td></td>
</tr>
<tr>
<td>sraq[.]</td>
<td>Shift Right with MQ</td>
<td>X</td>
<td>31</td>
<td>664</td>
<td></td>
</tr>
<tr>
<td>st</td>
<td>Store</td>
<td>D</td>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stb</td>
<td>Store Byte</td>
<td>D</td>
<td>38</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stbrx</td>
<td>Store Byte-Reversed Indexed</td>
<td>X</td>
<td>31</td>
<td>662</td>
<td></td>
</tr>
<tr>
<td>stbu</td>
<td>Store Byte with Update</td>
<td>D</td>
<td>39</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stbux</td>
<td>Store Byte with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>247</td>
<td></td>
</tr>
<tr>
<td>stbx</td>
<td>Store Byte Indexed</td>
<td>X</td>
<td>31</td>
<td>215</td>
<td></td>
</tr>
<tr>
<td>stfd</td>
<td>Store Floating-Point Double</td>
<td>D</td>
<td>54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stfdx</td>
<td>Store Floating-Point Double with Update</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stfd ux</td>
<td>Store Floating-Point Double with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>759</td>
<td></td>
</tr>
<tr>
<td>stfd x</td>
<td>Store Floating-Point Double Indexed</td>
<td>X</td>
<td>31</td>
<td>727</td>
<td></td>
</tr>
<tr>
<td>stfq</td>
<td>Store Floating-Point Quad</td>
<td></td>
<td>POWER2</td>
<td>DS</td>
<td>60</td>
</tr>
<tr>
<td>stfq x</td>
<td>Store Floating-Point Quad with Update Indexed</td>
<td></td>
<td>POWER2</td>
<td>DS</td>
<td>61</td>
</tr>
<tr>
<td>stfq ux</td>
<td>Store Floating-Point Quad with Update Indexed</td>
<td></td>
<td>POWER2</td>
<td>X</td>
<td>951</td>
</tr>
<tr>
<td>stfq x</td>
<td>Store Floating-Point Quad Indexed</td>
<td></td>
<td>POWER2</td>
<td>X</td>
<td>919</td>
</tr>
<tr>
<td>stfs</td>
<td>Store Floating-Point Single</td>
<td></td>
<td>D</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>stfs u</td>
<td>Store Floating-Point Single with Update</td>
<td></td>
<td>D</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Mode</td>
<td>Reg</td>
<td>addr</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------</td>
<td>------</td>
<td>-----</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>stfsux</td>
<td>Store Floating-Point Single with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>695</td>
<td></td>
</tr>
<tr>
<td>stfsx</td>
<td>Store Floating-Point Single Indexed</td>
<td>X</td>
<td>31</td>
<td>663</td>
<td></td>
</tr>
<tr>
<td>sth</td>
<td>Store Half</td>
<td>D</td>
<td>44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sthbx</td>
<td>Store Half Byte-Reverse Indexed</td>
<td>X</td>
<td>31</td>
<td>918</td>
<td></td>
</tr>
<tr>
<td>sthu</td>
<td>Store Half with Update</td>
<td>D</td>
<td>45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sthux</td>
<td>Store Half with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>439</td>
<td></td>
</tr>
<tr>
<td>sthx</td>
<td>Store Half Indexed</td>
<td>X</td>
<td>31</td>
<td>407</td>
<td></td>
</tr>
<tr>
<td>stm</td>
<td>Store Multiple</td>
<td>D</td>
<td>47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stsi</td>
<td>Store String Immediate</td>
<td>X</td>
<td>31</td>
<td>725</td>
<td></td>
</tr>
<tr>
<td>stsx</td>
<td>Store String Indexed</td>
<td>X</td>
<td>31</td>
<td>661</td>
<td></td>
</tr>
<tr>
<td>stu</td>
<td>Store with Update</td>
<td>D</td>
<td>37</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stux</td>
<td>Store with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td>stx</td>
<td>Store Indexed</td>
<td>X</td>
<td>31</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>svc</td>
<td>Supervisor Call</td>
<td>SC</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>Trap</td>
<td>X</td>
<td>31</td>
<td>04</td>
<td></td>
</tr>
<tr>
<td>ti</td>
<td>Trap Immediate</td>
<td>D</td>
<td>03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tlbi</td>
<td>Translation Look-aside Buffer Invalidate Entry</td>
<td>X</td>
<td>31</td>
<td>306</td>
<td></td>
</tr>
<tr>
<td>xor[]</td>
<td>XOR</td>
<td>X</td>
<td>31</td>
<td>316</td>
<td></td>
</tr>
<tr>
<td>xoril</td>
<td>XOR Immediate Lower</td>
<td>D</td>
<td>26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xoriu</td>
<td>XOR Immediate Upper</td>
<td>D</td>
<td>27</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Appendix F. PowerPC Instructions

### Table 37. PowerPC Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Format</th>
<th>Primary Op Code</th>
<th>Extended Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>add[]</td>
<td>Add</td>
<td>XO</td>
<td>31</td>
<td>266</td>
</tr>
<tr>
<td>addc[]</td>
<td>Add Carrying</td>
<td>XO</td>
<td>31</td>
<td>10</td>
</tr>
<tr>
<td>adde[]</td>
<td>Add Extended</td>
<td>XO</td>
<td>31</td>
<td>138</td>
</tr>
<tr>
<td>addi</td>
<td>Add Immediate</td>
<td>D</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>addic</td>
<td>Add Immediate Carrying</td>
<td>D</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>addic.</td>
<td>Add Immediate Carrying</td>
<td>D</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>addis</td>
<td>Add Immediate Shifted</td>
<td>D</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>addme[]</td>
<td>Add to Minus One Extended</td>
<td>XO</td>
<td>31</td>
<td>234</td>
</tr>
<tr>
<td>addze[]</td>
<td>Add to Zero Extended</td>
<td>XO</td>
<td>31</td>
<td>202</td>
</tr>
<tr>
<td>and[]</td>
<td>AND</td>
<td>X</td>
<td>31</td>
<td>28</td>
</tr>
<tr>
<td>andc[]</td>
<td>AND with Complement</td>
<td>X</td>
<td>31</td>
<td>60</td>
</tr>
<tr>
<td>andi.</td>
<td>AND Immediate Shifted</td>
<td>D</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>andis.</td>
<td>AND Immediate</td>
<td>D</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>b[]a</td>
<td>Branch</td>
<td>I</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>bc[]a</td>
<td>Branch Conditional</td>
<td>B</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>bcctr[l]</td>
<td>Branch Conditional to</td>
<td>XL</td>
<td>19</td>
<td>528</td>
</tr>
<tr>
<td></td>
<td>Count Register</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bclr[l]</td>
<td>Branch Conditional Link</td>
<td>XL</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Register</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp</td>
<td>Compare</td>
<td>X</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>cmpi</td>
<td>Compare Immediate</td>
<td>D</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>cmpl</td>
<td>Compare Logical</td>
<td>X</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>cmpli</td>
<td>Compare Logical</td>
<td>D</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>cntlzd</td>
<td>Count Leading Zeros</td>
<td>X</td>
<td>31</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cnttzw[]</td>
<td>Count Leading Zeros</td>
<td>X</td>
<td>31</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>crand</td>
<td>Condition Register</td>
<td>XL</td>
<td>19</td>
<td>257</td>
</tr>
<tr>
<td>crandc</td>
<td>Condition Register</td>
<td>XL</td>
<td>19</td>
<td>129</td>
</tr>
<tr>
<td></td>
<td>AND with Complement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>creqv</td>
<td>Condition Register</td>
<td>XL</td>
<td>19</td>
<td>289</td>
</tr>
<tr>
<td></td>
<td>Equivalent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>crnand</td>
<td>Condition Register</td>
<td>XL</td>
<td>19</td>
<td>225</td>
</tr>
<tr>
<td></td>
<td>NAND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Instruction</td>
<td>Format</td>
<td>Primary Op Code</td>
<td>Extended Op Code</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>--------</td>
<td>----------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>crnor</td>
<td>Condition Register NOR</td>
<td>XL</td>
<td>19</td>
<td>33</td>
</tr>
<tr>
<td>cror</td>
<td>Condition Register OR</td>
<td>XL</td>
<td>19</td>
<td>449</td>
</tr>
<tr>
<td>crorc</td>
<td>Condition Register OR with Complement</td>
<td>XL</td>
<td>19</td>
<td>417</td>
</tr>
<tr>
<td>crxor</td>
<td>Condition Register XOR</td>
<td>XL</td>
<td>19</td>
<td>193</td>
</tr>
<tr>
<td>dcbf</td>
<td>Data Cache Block Flush</td>
<td>X</td>
<td>31</td>
<td>86</td>
</tr>
<tr>
<td>dcbi</td>
<td>Data Cache Block Invalidate</td>
<td>X</td>
<td>31</td>
<td>470</td>
</tr>
<tr>
<td>dcbs</td>
<td>Data Cache Block Store</td>
<td>X</td>
<td>31</td>
<td>54</td>
</tr>
<tr>
<td>dcbt</td>
<td>Data Cache Block Touch</td>
<td>X</td>
<td>31</td>
<td>278</td>
</tr>
<tr>
<td>dcbtst</td>
<td>Data Cache Block Touch for Store</td>
<td>X</td>
<td>31</td>
<td>246</td>
</tr>
<tr>
<td>dcbz</td>
<td>Data Cache Block Set to Zero</td>
<td>X</td>
<td>31</td>
<td>1014</td>
</tr>
<tr>
<td>divd</td>
<td>Divide Doubleword</td>
<td>XO</td>
<td>31</td>
<td>489</td>
</tr>
<tr>
<td>divdu</td>
<td>Divide Doubleword Unsigned</td>
<td>XO</td>
<td>31</td>
<td>457</td>
</tr>
<tr>
<td>divw[]</td>
<td>Divide Word</td>
<td>XO</td>
<td>31</td>
<td>491</td>
</tr>
<tr>
<td>divwu[]</td>
<td>Divide Word Unsigned</td>
<td>XO</td>
<td>31</td>
<td>459</td>
</tr>
<tr>
<td>ecix</td>
<td>External Control in Word Indexed (opt.)</td>
<td>X</td>
<td>31</td>
<td>310</td>
</tr>
<tr>
<td>ecow</td>
<td>External Control out Word Indexed (opt.)</td>
<td>X</td>
<td>31</td>
<td>438</td>
</tr>
<tr>
<td>eieio</td>
<td>Enforce In-order Execution of I/O</td>
<td>X</td>
<td>31</td>
<td>854</td>
</tr>
<tr>
<td>eqv[]</td>
<td>Equivalent</td>
<td>X</td>
<td>31</td>
<td>284</td>
</tr>
<tr>
<td>exts[]</td>
<td>Extend Sign Byte</td>
<td>X</td>
<td>31</td>
<td>954</td>
</tr>
<tr>
<td>extsh[]</td>
<td>Extend Sign Halfword</td>
<td>XO</td>
<td>31</td>
<td>922</td>
</tr>
<tr>
<td>extsw</td>
<td>Extend Sign Word</td>
<td>X</td>
<td>31</td>
<td>986</td>
</tr>
<tr>
<td>fabs[]</td>
<td>Floating Absolute Value</td>
<td>X</td>
<td>63</td>
<td>264</td>
</tr>
<tr>
<td>fadd[]</td>
<td>Floating Add</td>
<td>A</td>
<td>63</td>
<td>21</td>
</tr>
<tr>
<td>fadds[]</td>
<td>Floating Add Single</td>
<td>A</td>
<td>59</td>
<td>21</td>
</tr>
<tr>
<td>fcfid</td>
<td>Floating Convert from Integer Doubleword</td>
<td>X</td>
<td>63</td>
<td>846</td>
</tr>
<tr>
<td>fcmpo</td>
<td>Floating Compare Ordered</td>
<td>X</td>
<td>63</td>
<td>32</td>
</tr>
<tr>
<td>fcmpu</td>
<td>Floating Compare Unordered</td>
<td>XL</td>
<td>63</td>
<td>0</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Instruction</td>
<td>Format</td>
<td>Primary Op Code</td>
<td>Extended Op Code</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>--------</td>
<td>-----------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>fctid</td>
<td>Floating Convert to Integer Doubleword</td>
<td>X 63</td>
<td>814</td>
<td></td>
</tr>
<tr>
<td>fctidz</td>
<td>Floating Convert to Integer Doubleword with Round Toward Zero</td>
<td>X 63</td>
<td>815</td>
<td></td>
</tr>
<tr>
<td>fctiw[,]</td>
<td>Floating Convert to Integer Word</td>
<td>X 63</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>fctiwz[,]</td>
<td>Floating Convert to Integer Word with Round to Zero</td>
<td>XL 63</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>fdiv[,]</td>
<td>Floating Divide</td>
<td>A 63</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>fdivs[,]</td>
<td>Floating Divide Single</td>
<td>A 59</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>fmadd[,]</td>
<td>Floating Multiply-Add</td>
<td>A 63</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>fnmadd[,]</td>
<td>Floating Multiply-Add Single</td>
<td>A 59</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>fmr[,]</td>
<td>Floating Move Register</td>
<td>X 63</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>fmsub[,]</td>
<td>Floating Multiply-Subtract</td>
<td>A 63</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>fnmsub[,]</td>
<td>Floating Multiply-Subtract Single</td>
<td>A 59</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>fnmul[,]</td>
<td>Floating Multiply</td>
<td>A 63</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>fnmuls[,]</td>
<td>Floating Multiply Single</td>
<td>A 59</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>fnneg[,]</td>
<td>Floating Negate</td>
<td>X 63</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>fnmadd[,]</td>
<td>Floating Negative Multiply-Add</td>
<td>A 63</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>fnmadds[,]</td>
<td>Floating Negative Multiply-Add Single</td>
<td>A 59</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>fnmsub[,]</td>
<td>Floating Negative Multiply-Subtract</td>
<td>A 63</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>fnmsubs[,]</td>
<td>Floating Negative Multiply-Subtract Single</td>
<td>A 59</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>fnres[,]</td>
<td>Floating Reciprocal Estimate Single (optional)</td>
<td>A 59</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>fnrsp[,]</td>
<td>Floating Round to Single Precision</td>
<td>X 63</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>fnrsqrte[,]</td>
<td>Floating Reciprocal Square Root Estimate (optional)</td>
<td>A 63</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>fnfse[,]</td>
<td>Floating-Point Select (optional)</td>
<td>A 63</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Instruction</td>
<td>Format</td>
<td>Primary Op Code</td>
<td>Extended Op Code</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>--------</td>
<td>----------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>fsub[.]</td>
<td>Floating Subtract</td>
<td>A</td>
<td>63</td>
<td>20</td>
</tr>
<tr>
<td>fsubs[.]</td>
<td>Floating Subtract Single</td>
<td>A</td>
<td>59</td>
<td>20</td>
</tr>
<tr>
<td>icbi</td>
<td>Instruction Cache Block Invalidate</td>
<td>X</td>
<td>31</td>
<td>982</td>
</tr>
<tr>
<td>isync</td>
<td>Instruction Synchronize</td>
<td>X</td>
<td>19</td>
<td>150</td>
</tr>
<tr>
<td>lbz</td>
<td>Load Byte and Zero</td>
<td>D</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>lbzu</td>
<td>Load Byte and Zero with Update</td>
<td>D</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>lbzux</td>
<td>Load Byte and Zero with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>119</td>
</tr>
<tr>
<td>lbzx</td>
<td>Load Byte and Zero Indexed</td>
<td>X</td>
<td>31</td>
<td>87</td>
</tr>
<tr>
<td>ld</td>
<td>Load Doubleword</td>
<td>DS</td>
<td>58</td>
<td>0</td>
</tr>
<tr>
<td>ldarx</td>
<td>Load Doubleword and Reserve Indexed</td>
<td>X</td>
<td>31</td>
<td>84</td>
</tr>
<tr>
<td>ldru</td>
<td>Load Doubleword with Update</td>
<td>DS</td>
<td>58</td>
<td>1</td>
</tr>
<tr>
<td>ldux</td>
<td>Load Doubleword with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>53</td>
</tr>
<tr>
<td>ldx</td>
<td>Load Doubleword Indexed</td>
<td>X</td>
<td>31</td>
<td>21</td>
</tr>
<tr>
<td>lfd</td>
<td>Load Floating-Point Double</td>
<td>D</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>lfdru</td>
<td>Load Floating-Point Double with Update</td>
<td>D</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>lfdux</td>
<td>Load Floating-Point Double with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>631</td>
</tr>
<tr>
<td>lfdx</td>
<td>Load Floating-Point Double Indexed</td>
<td>X</td>
<td>31</td>
<td>599</td>
</tr>
<tr>
<td>lfs</td>
<td>Load Floating-Point Single</td>
<td>D</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>lfsu</td>
<td>Load Floating-Point Single with Update</td>
<td>D</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>lfsux</td>
<td>Load Floating-Point Single with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>567</td>
</tr>
<tr>
<td>lfsx</td>
<td>Load Floating-Point Single Indexed</td>
<td>X</td>
<td>31</td>
<td>535</td>
</tr>
<tr>
<td>lha</td>
<td>Load Half Algebraic</td>
<td>D</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>lhau</td>
<td>Load Half Algebraic with Update</td>
<td>D</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>lhaux</td>
<td>Load Half Algebraic with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>375</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Instruction</td>
<td>Format</td>
<td>Primary Op Code</td>
<td>Extended Op Code</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>--------</td>
<td>-----------------</td>
<td>------------------</td>
</tr>
<tr>
<td>lhax</td>
<td>Load Half Algebraic Indexed</td>
<td>X</td>
<td>31</td>
<td>343</td>
</tr>
<tr>
<td>lhbrx</td>
<td>Load Half Byte-Reversed Indexed</td>
<td>X</td>
<td>31</td>
<td>790</td>
</tr>
<tr>
<td>lhz</td>
<td>Load Half and Zero Indexed</td>
<td>D</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>lhzu</td>
<td>Load Half and Zero with Update</td>
<td>D</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>lhzux</td>
<td>Load Half and Zero with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>331</td>
</tr>
<tr>
<td>lhzx</td>
<td>Load Half and Zero Indexed</td>
<td>X</td>
<td>31</td>
<td>279</td>
</tr>
<tr>
<td>lmw</td>
<td>Load Multiple Word Indexed</td>
<td>D</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>lswi</td>
<td>Load String Word Immediate</td>
<td>X</td>
<td>31</td>
<td>597</td>
</tr>
<tr>
<td>lswx</td>
<td>Load String Word Indexed</td>
<td>X</td>
<td>31</td>
<td>533</td>
</tr>
<tr>
<td>lwa</td>
<td>Load Word Algebraic</td>
<td>DS</td>
<td>58</td>
<td>2</td>
</tr>
<tr>
<td>lwarx</td>
<td>Load Word and Reserve Indexed</td>
<td>X</td>
<td>31</td>
<td>20</td>
</tr>
<tr>
<td>lwaux</td>
<td>Load Word Algebraic with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>373</td>
</tr>
<tr>
<td>lwax</td>
<td>Load Word Algebraic Indexed</td>
<td>X</td>
<td>31</td>
<td>341</td>
</tr>
<tr>
<td>lwbrx</td>
<td>Load Word Byte-Reversed Indexed</td>
<td>X</td>
<td>31</td>
<td>534</td>
</tr>
<tr>
<td>lwz</td>
<td>Load Word and Zero Indexed</td>
<td>D</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>lwzu</td>
<td>Load Word with Zero Update</td>
<td>D</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>lwzux</td>
<td>Load Word and Zero with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>55</td>
</tr>
<tr>
<td>lwzx</td>
<td>Load Word and Zero Indexed</td>
<td>X</td>
<td>31</td>
<td>23</td>
</tr>
<tr>
<td>mcrf</td>
<td>Move Condition Register Field</td>
<td>XL</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>mcrfs</td>
<td>Move to Condition Register from FPSCR</td>
<td>X</td>
<td>63</td>
<td>64</td>
</tr>
<tr>
<td>mcrxr</td>
<td>Move to Condition Register from XER</td>
<td>X</td>
<td>31</td>
<td>512</td>
</tr>
<tr>
<td>mfcr</td>
<td>Move from Condition Register</td>
<td>X</td>
<td>31</td>
<td>19</td>
</tr>
<tr>
<td>mffs[.]</td>
<td>Move from FPSCR</td>
<td>X</td>
<td>63</td>
<td>583</td>
</tr>
<tr>
<td>mfmsr</td>
<td>Move from Machine State Register</td>
<td>X</td>
<td>31</td>
<td>83</td>
</tr>
</tbody>
</table>
### Table 37. PowerPC Instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Format</th>
<th>Primary Op Code</th>
<th>Extended Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>mfspr</td>
<td>Move from Special-Purpose Register</td>
<td>X</td>
<td>31</td>
<td>339</td>
</tr>
<tr>
<td>mfsr</td>
<td>Move from Segment Register</td>
<td>X</td>
<td>31</td>
<td>595</td>
</tr>
<tr>
<td>mfsrin</td>
<td>Move from Segment Register Indirect</td>
<td>X</td>
<td>31</td>
<td>659</td>
</tr>
<tr>
<td>mtcrf</td>
<td>Move to Condition Register Fields</td>
<td>XFX</td>
<td>31</td>
<td>144</td>
</tr>
<tr>
<td>mtfsb0[,]</td>
<td>Move to FPSCR Bit 0</td>
<td>X</td>
<td>63</td>
<td>70</td>
</tr>
<tr>
<td>mtfsb1[,]</td>
<td>Move to FPSCR Bit 1</td>
<td>X</td>
<td>63</td>
<td>38</td>
</tr>
<tr>
<td>mtfsf[,]</td>
<td>Move to FPSCR Fields</td>
<td>XFL</td>
<td>63</td>
<td>711</td>
</tr>
<tr>
<td>mtfsfi[,]</td>
<td>Move to FPSCR Field Immediate</td>
<td>X</td>
<td>63</td>
<td>134</td>
</tr>
<tr>
<td>mtmsr</td>
<td>Move to Machine State Register</td>
<td>X</td>
<td>31</td>
<td>146</td>
</tr>
<tr>
<td>mtspri</td>
<td>Move to Special-Purpose Register</td>
<td>X</td>
<td>31</td>
<td>467</td>
</tr>
<tr>
<td>mtspri</td>
<td>Move to Segment Register</td>
<td>X</td>
<td>31</td>
<td>210</td>
</tr>
<tr>
<td>mtspri</td>
<td>Move to Segment Register Indirect</td>
<td>X</td>
<td>31</td>
<td>242</td>
</tr>
<tr>
<td>mulhd</td>
<td>Multiply High Doubleword</td>
<td>XO</td>
<td>31</td>
<td>73</td>
</tr>
<tr>
<td>mulhdu</td>
<td>Multiply High Doubleword Unsigned</td>
<td>XO</td>
<td>31</td>
<td>9</td>
</tr>
<tr>
<td>mulhw[,]</td>
<td>Multiply High Word</td>
<td>XO</td>
<td>31</td>
<td>75</td>
</tr>
<tr>
<td>mulhwu[,]</td>
<td>Multiply High Word Unsigned</td>
<td>XO</td>
<td>31</td>
<td>11</td>
</tr>
<tr>
<td>mulld</td>
<td>Multiply Low Doubleword</td>
<td>XO</td>
<td>31</td>
<td>233</td>
</tr>
<tr>
<td>mulld</td>
<td>Multiply Low Immediate</td>
<td>D</td>
<td>07</td>
<td></td>
</tr>
<tr>
<td>mulw[o][.</td>
<td>Multiply Low Word</td>
<td>XO</td>
<td>31</td>
<td>235</td>
</tr>
<tr>
<td>nand[,]</td>
<td>NAND</td>
<td>X</td>
<td>31</td>
<td>476</td>
</tr>
<tr>
<td>neg[o][.</td>
<td>Negate</td>
<td>XO</td>
<td>31</td>
<td>104</td>
</tr>
<tr>
<td>nor[,]</td>
<td>NOR</td>
<td>X</td>
<td>31</td>
<td>124</td>
</tr>
<tr>
<td>or[,]</td>
<td>OR</td>
<td>X</td>
<td>31</td>
<td>444</td>
</tr>
<tr>
<td>orc[,]</td>
<td>OR with Complement</td>
<td>X</td>
<td>31</td>
<td>412</td>
</tr>
<tr>
<td>ori</td>
<td>OR Immediate</td>
<td>D</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>oris</td>
<td>OR Immediate Shifted</td>
<td>D</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>rfi</td>
<td>Return from Interrupt</td>
<td>X</td>
<td>19</td>
<td>50</td>
</tr>
</tbody>
</table>
Table 37. PowerPC Instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Format</th>
<th>Primary Op Code</th>
<th>Extended Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>rldcl</td>
<td>Rotate Left Doubleword then Clear Left</td>
<td>MDS</td>
<td>30</td>
<td>8</td>
</tr>
<tr>
<td>rldcr</td>
<td>Rotate Left Doubleword then Clear Right</td>
<td>MDS</td>
<td>30</td>
<td>9</td>
</tr>
<tr>
<td>rldic</td>
<td>Rotate Left Doubleword Immediate then Clear</td>
<td>MD</td>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>rldicl</td>
<td>Rotate Left Doubleword Immediate then Clear Left</td>
<td>MD</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>rldicr</td>
<td>Rotate Left Doubleword Immediate then Clear Right</td>
<td>MD</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>rldimi</td>
<td>Rotate Left Doubleword Immediate then Mask Insert</td>
<td>MD</td>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>rlwimi[].</td>
<td>Rotate Left Word Immediate then Mask Insert</td>
<td>M</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>rlwinm[].</td>
<td>Rotate Left Word Immediate then AND with Mask</td>
<td>M</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>rlwnm[].</td>
<td>Rotate Left Word then AND with Mask</td>
<td>M</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td>System Call</td>
<td>SC</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>si</td>
<td>Subtract Immediate</td>
<td>D</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>si.</td>
<td>Subtract Immediate and Record</td>
<td>D</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>slbia</td>
<td>SLB Invalidate All</td>
<td>X</td>
<td>31</td>
<td>498</td>
</tr>
<tr>
<td>slbie</td>
<td>SLB Invalidate Entry</td>
<td>X</td>
<td>31</td>
<td>434</td>
</tr>
<tr>
<td>sld</td>
<td>Shift Left Doubleword</td>
<td>X</td>
<td>31</td>
<td>27</td>
</tr>
<tr>
<td>slw[]</td>
<td>Shift Left Word</td>
<td>X</td>
<td>31</td>
<td>24</td>
</tr>
<tr>
<td>srad</td>
<td>Shift Right Algebraic Doubleword</td>
<td>X</td>
<td>31</td>
<td>794</td>
</tr>
<tr>
<td>sradi</td>
<td>Shift Right Algebraic Doubleword Immediate</td>
<td>XS</td>
<td>31</td>
<td>413</td>
</tr>
<tr>
<td>srd</td>
<td>Shift Right Doubleword</td>
<td>X</td>
<td>31</td>
<td>539</td>
</tr>
<tr>
<td>sraw[]</td>
<td>Shift Right Algebraic Word</td>
<td>X</td>
<td>31</td>
<td>792</td>
</tr>
<tr>
<td>srawi[]</td>
<td>Shift Right Algebraic Word Immediate</td>
<td>X</td>
<td>31</td>
<td>824</td>
</tr>
</tbody>
</table>
Table 37. PowerPC Instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Format</th>
<th>Primary Op Code</th>
<th>Extended Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>srw[,]</td>
<td>Shift Right Word</td>
<td>X</td>
<td>31</td>
<td>536</td>
</tr>
<tr>
<td>stb</td>
<td>Store Byte</td>
<td>D</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>stbu</td>
<td>Store Byte with Update</td>
<td>D</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>stbux</td>
<td>Store Byte with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>247</td>
</tr>
<tr>
<td>stbx</td>
<td>Store Byte Indexed</td>
<td>X</td>
<td>31</td>
<td>215</td>
</tr>
<tr>
<td>std</td>
<td>Store Doubleword</td>
<td>DS</td>
<td>62</td>
<td>0</td>
</tr>
<tr>
<td>stdcx</td>
<td>Store Doubleword Conditional Indexed</td>
<td>X</td>
<td>31</td>
<td>214</td>
</tr>
<tr>
<td>stdu</td>
<td>Store Doubleword with Update</td>
<td>DS</td>
<td>62</td>
<td>1</td>
</tr>
<tr>
<td>stdux</td>
<td>Store Doubleword with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>181</td>
</tr>
<tr>
<td>stdx</td>
<td>Store Doubleword Indexed</td>
<td>X</td>
<td>31</td>
<td>149</td>
</tr>
<tr>
<td>stdf</td>
<td>Store Floating-Point Double</td>
<td>D</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>stdfu</td>
<td>Store Floating-Point Double with Update</td>
<td>D</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>stdfux</td>
<td>Store Floating-Point Double with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>759</td>
</tr>
<tr>
<td>stdfx</td>
<td>Store Floating-Point Double Indexed</td>
<td>X</td>
<td>31</td>
<td>727</td>
</tr>
<tr>
<td>stfiwx</td>
<td>Store Floating-Point as Integer Word Indexed (optional)</td>
<td>X</td>
<td>31</td>
<td>983</td>
</tr>
<tr>
<td>stfs</td>
<td>Store Floating-Point Single</td>
<td>D</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>stfsu</td>
<td>Store Floating-Point Single with Update</td>
<td>D</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>stfsux</td>
<td>Store Floating-Point Single with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>695</td>
</tr>
<tr>
<td>stfsx</td>
<td>Store Floating-Point Single Indexed</td>
<td>X</td>
<td>31</td>
<td>663</td>
</tr>
<tr>
<td>sth</td>
<td>Store Half</td>
<td>D</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>sthbrx</td>
<td>Store Half Byte-Reverse Indexed</td>
<td>X</td>
<td>31</td>
<td>918</td>
</tr>
<tr>
<td>sthu</td>
<td>Store Half with Update</td>
<td>D</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>sthux</td>
<td>Store Half with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>439</td>
</tr>
<tr>
<td>sthx</td>
<td>Store Half Indexed</td>
<td>X</td>
<td>31</td>
<td>407</td>
</tr>
<tr>
<td>stmw</td>
<td>Store Multiple Word</td>
<td>D</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Instruction</td>
<td>Format</td>
<td>Primary Op Code</td>
<td>Extended Op Code</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------------------</td>
<td>--------</td>
<td>-----------------</td>
<td>------------------</td>
</tr>
<tr>
<td>stswi</td>
<td>Store String Word Immediate</td>
<td>X</td>
<td>31</td>
<td>725</td>
</tr>
<tr>
<td>stswx</td>
<td>Store String Word Indexed</td>
<td>X</td>
<td>31</td>
<td>661</td>
</tr>
<tr>
<td>stw</td>
<td>Store</td>
<td>D</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>stwbrx</td>
<td>Store Word Byte-Reversed Indexed</td>
<td>X</td>
<td>31</td>
<td>662</td>
</tr>
<tr>
<td>stwcx.</td>
<td>Store Word Conditional Indexed</td>
<td>X</td>
<td>31</td>
<td>150</td>
</tr>
<tr>
<td>stwu</td>
<td>Store Word with Update</td>
<td>D</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>stwux</td>
<td>Store Word with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>183</td>
</tr>
<tr>
<td>stwx</td>
<td>Store Word Indexed</td>
<td>X</td>
<td>31</td>
<td>151</td>
</tr>
<tr>
<td>sub[o][]</td>
<td>Subtract from</td>
<td>XO</td>
<td>31</td>
<td>40</td>
</tr>
<tr>
<td>subfc[o][]</td>
<td>Subtract from Carrying</td>
<td>XO</td>
<td>31</td>
<td>08</td>
</tr>
<tr>
<td>subfe[o][]</td>
<td>Subtract from Extended</td>
<td>XO</td>
<td>31</td>
<td>136</td>
</tr>
<tr>
<td>subfic</td>
<td>Subtract from Immediate Carrying</td>
<td>D</td>
<td>08</td>
<td></td>
</tr>
<tr>
<td>subfme[o][]</td>
<td>Subtract from Minus One Extended</td>
<td>XO</td>
<td>31</td>
<td>232</td>
</tr>
<tr>
<td>subfze[o][]</td>
<td>Subtract from Zero Extended</td>
<td>XO</td>
<td>31</td>
<td>200</td>
</tr>
<tr>
<td>sync</td>
<td>Synchronize</td>
<td>X</td>
<td>31</td>
<td>598</td>
</tr>
<tr>
<td>td</td>
<td>Trap Doubleword</td>
<td>X</td>
<td>31</td>
<td>68</td>
</tr>
<tr>
<td>tdi</td>
<td>Trap Doubleword Immediate</td>
<td>D</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>tlbie</td>
<td>Translation Look-aside Buffer Invalidate Entry (optional)</td>
<td>X</td>
<td>31</td>
<td>306</td>
</tr>
<tr>
<td>tlbsync</td>
<td>Translation Look-aside Buffer Synchronize (optional)</td>
<td>X</td>
<td>31</td>
<td>566</td>
</tr>
<tr>
<td>tw</td>
<td>Trap Word</td>
<td>X</td>
<td>31</td>
<td>04</td>
</tr>
<tr>
<td>twi</td>
<td>Trap Word Immediate</td>
<td>D</td>
<td>03</td>
<td></td>
</tr>
<tr>
<td>xor[]</td>
<td>XOR</td>
<td>X</td>
<td>31</td>
<td>316</td>
</tr>
<tr>
<td>xor</td>
<td>XOR Immediate</td>
<td>D</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>xoris</td>
<td>XOR Immediate Shift</td>
<td>D</td>
<td>27</td>
<td></td>
</tr>
</tbody>
</table>
## Appendix G. PowerPC 601 RISC Microprocessor Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Format</th>
<th>Primary Op Code</th>
<th>Extended Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[o][.]</td>
<td>Add Carrying</td>
<td>XO</td>
<td>31</td>
<td>10</td>
</tr>
<tr>
<td>abs[o][.]</td>
<td>Absolute</td>
<td>XO</td>
<td>31</td>
<td>360</td>
</tr>
<tr>
<td>add[o][.]</td>
<td>Add</td>
<td>XO</td>
<td>31</td>
<td>266</td>
</tr>
<tr>
<td>addc[o][.]</td>
<td>Add Carrying</td>
<td>XO</td>
<td>31</td>
<td>10</td>
</tr>
<tr>
<td>ade[o][.]</td>
<td>Add Extended</td>
<td>XO</td>
<td>31</td>
<td>138</td>
</tr>
<tr>
<td>addi</td>
<td>Add Immediate</td>
<td>D</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>addic</td>
<td>Add Immediate Carrying</td>
<td>D</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>addic.</td>
<td>Add Immediate Carrying and Record</td>
<td>D</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>addis</td>
<td>Add Immediate Shifted</td>
<td>D</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>addme[o][.]</td>
<td>Add to Minus One Extended</td>
<td>XO</td>
<td>31</td>
<td>234</td>
</tr>
<tr>
<td>addze[o][.]</td>
<td>Add to Zero Extended</td>
<td>XO</td>
<td>31</td>
<td>202</td>
</tr>
<tr>
<td>ae[o][.]</td>
<td>Add Extended</td>
<td>XO</td>
<td>31</td>
<td>138</td>
</tr>
<tr>
<td>ai</td>
<td>Add Immediate</td>
<td>D</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>ai.</td>
<td>Add Immediate and Record</td>
<td>D</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>ame[o][.]</td>
<td>Add to Minus One Extended</td>
<td>XO</td>
<td>31</td>
<td>234</td>
</tr>
<tr>
<td>and[.]</td>
<td>AND</td>
<td>X</td>
<td>31</td>
<td>28</td>
</tr>
<tr>
<td>andc[.]</td>
<td>AND with Complement</td>
<td>X</td>
<td>31</td>
<td>60</td>
</tr>
<tr>
<td>andi.</td>
<td>AND Immediate</td>
<td>D</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>andil.</td>
<td>AND Immediate Lower</td>
<td>D</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>andis.</td>
<td>AND Immediate Shifted</td>
<td>D</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>andiu.</td>
<td>AND Immediate Upper</td>
<td>D</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>aze[o][.]</td>
<td>Add to Zero Extended</td>
<td>XO</td>
<td>31</td>
<td>202</td>
</tr>
<tr>
<td>b[i][a]</td>
<td>Branch</td>
<td>I</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>bc[i][a]</td>
<td>Branch Conditional</td>
<td>B</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>bcc[l]</td>
<td>Branch Conditional to Count Register</td>
<td>XL</td>
<td>19</td>
<td>528</td>
</tr>
<tr>
<td>bcctr[l]</td>
<td>Branch Conditional to Count Register</td>
<td>XL</td>
<td>19</td>
<td>528</td>
</tr>
<tr>
<td>bclr[l]</td>
<td>Branch Conditional Link Register</td>
<td>XL</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td>bcr[l]</td>
<td>Branch Conditional Register</td>
<td>XL</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
<td>Size</td>
<td>Word/Byte</td>
<td>Bits</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
<td>-----------</td>
<td>------</td>
</tr>
<tr>
<td>cal</td>
<td>Compute Address Lower</td>
<td>D</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>cau</td>
<td>Compute Address Upper</td>
<td>D</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>cax[]</td>
<td>Compute Address</td>
<td>XO</td>
<td>31</td>
<td>266</td>
</tr>
<tr>
<td>clcs</td>
<td>Cache Line Compute Size</td>
<td>X</td>
<td>31</td>
<td>531</td>
</tr>
<tr>
<td>cmp</td>
<td>Compare</td>
<td>X</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>cmpi</td>
<td>Compare Immediate</td>
<td>D</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>cmpl</td>
<td>Compare Logical</td>
<td>X</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>cmpli</td>
<td>Compare Logical Immediate</td>
<td>D</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>cntlz[]</td>
<td>Count Leading Zeros</td>
<td>X</td>
<td>31</td>
<td>26</td>
</tr>
<tr>
<td>cntlzw[]</td>
<td>Count Leading Zeros Word</td>
<td>X</td>
<td>31</td>
<td>26</td>
</tr>
<tr>
<td>crand</td>
<td>Condition Register AND</td>
<td>XL</td>
<td>19</td>
<td>257</td>
</tr>
<tr>
<td>crandc</td>
<td>Condition Register AND with Complement</td>
<td>XL</td>
<td>19</td>
<td>129</td>
</tr>
<tr>
<td>creqv</td>
<td>Condition Register Equivalent</td>
<td>XL</td>
<td>19</td>
<td>289</td>
</tr>
<tr>
<td>crnand</td>
<td>Condition Register NAND</td>
<td>XL</td>
<td>19</td>
<td>225</td>
</tr>
<tr>
<td>crnor</td>
<td>Condition Register NOR</td>
<td>XL</td>
<td>19</td>
<td>33</td>
</tr>
<tr>
<td>cror</td>
<td>Condition Register OR</td>
<td>XL</td>
<td>19</td>
<td>449</td>
</tr>
<tr>
<td>croc</td>
<td>Condition Register OR with Complement</td>
<td>XL</td>
<td>19</td>
<td>417</td>
</tr>
<tr>
<td>croc</td>
<td>Condition Register XOR</td>
<td>XL</td>
<td>19</td>
<td>193</td>
</tr>
<tr>
<td>dcbf</td>
<td>Data Cache Block Flush</td>
<td>X</td>
<td>31</td>
<td>86</td>
</tr>
<tr>
<td>dcbi</td>
<td>Data Cache Block Invalidate</td>
<td>X</td>
<td>31</td>
<td>470</td>
</tr>
<tr>
<td>dcbst</td>
<td>Data Cache Block Store</td>
<td>X</td>
<td>31</td>
<td>54</td>
</tr>
<tr>
<td>dbt</td>
<td>Data Cache Block Touch</td>
<td>X</td>
<td>31</td>
<td>278</td>
</tr>
<tr>
<td>dbtst</td>
<td>Data Cache Block Touch for Store</td>
<td>X</td>
<td>31</td>
<td>246</td>
</tr>
<tr>
<td>dcbz</td>
<td>Data Cache Block Set to Zero</td>
<td>X</td>
<td>31</td>
<td>1014</td>
</tr>
<tr>
<td>dcs</td>
<td>Data Cache Synchronize</td>
<td>X</td>
<td>31</td>
<td>598</td>
</tr>
<tr>
<td>div[0][]</td>
<td>Divide</td>
<td>XO</td>
<td>31</td>
<td>331</td>
</tr>
<tr>
<td>divs[0][]</td>
<td>Divide Short</td>
<td>XO</td>
<td>31</td>
<td>363</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Mode</td>
<td>Size</td>
<td>Opcode</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------------------</td>
<td>------</td>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td>divw[0][ ]</td>
<td>Divide Word</td>
<td>XO</td>
<td>31</td>
<td>491</td>
</tr>
<tr>
<td>divwu[0][ ]</td>
<td>Divide Word Signed</td>
<td>XO</td>
<td>31</td>
<td>459</td>
</tr>
<tr>
<td>doz[0][ ]</td>
<td>Difference or Zero</td>
<td>XO</td>
<td>31</td>
<td>264</td>
</tr>
<tr>
<td>dozi</td>
<td>Difference or Zero Immediate</td>
<td>D</td>
<td>09</td>
<td></td>
</tr>
<tr>
<td>eciwx</td>
<td>External Control in Word Indexed</td>
<td>X</td>
<td>31</td>
<td>310</td>
</tr>
<tr>
<td>ecowx</td>
<td>External Control out Word Indexed</td>
<td>X</td>
<td>31</td>
<td>438</td>
</tr>
<tr>
<td>eieio</td>
<td>Enforce In-order Execution of I/O</td>
<td>X</td>
<td>31</td>
<td>854</td>
</tr>
<tr>
<td>eqv[ ]</td>
<td>Equivalent</td>
<td>X</td>
<td>31</td>
<td>284</td>
</tr>
<tr>
<td>exts[ ]</td>
<td>Extend Sign</td>
<td>X</td>
<td>31</td>
<td>922</td>
</tr>
<tr>
<td>extsb[ ]</td>
<td>Extend Sign Byte</td>
<td>X</td>
<td>31</td>
<td>954</td>
</tr>
<tr>
<td>extsh[ ]</td>
<td>Extend Sign Halfword</td>
<td>XO</td>
<td>31</td>
<td>922</td>
</tr>
<tr>
<td>fa[ ]</td>
<td>Floating Add</td>
<td>A</td>
<td>63</td>
<td>21</td>
</tr>
<tr>
<td>fabs[ ]</td>
<td>Floating Absolute Value</td>
<td>X</td>
<td>63</td>
<td>264</td>
</tr>
<tr>
<td>fadd[ ]</td>
<td>Floating Add</td>
<td>A</td>
<td>63</td>
<td>21</td>
</tr>
<tr>
<td>fadds[ ]</td>
<td>Floating Add Single</td>
<td>A</td>
<td>59</td>
<td>21</td>
</tr>
<tr>
<td>fcir[ ]</td>
<td>Floating Convert to Integer Word</td>
<td>X</td>
<td>63</td>
<td>14</td>
</tr>
<tr>
<td>fcirz[ ]</td>
<td>Floating Convert to Integer Word with Round to Zero</td>
<td>X</td>
<td>63</td>
<td>15</td>
</tr>
<tr>
<td>fcmpo</td>
<td>Floating Compare Ordered</td>
<td>X</td>
<td>63</td>
<td>32</td>
</tr>
<tr>
<td>fcmpu</td>
<td>Floating Compare Unordered</td>
<td>XL</td>
<td>63</td>
<td>0</td>
</tr>
<tr>
<td>fctiw[ ]</td>
<td>Floating Convert to Integer Word</td>
<td>X</td>
<td>63</td>
<td>14</td>
</tr>
<tr>
<td>fctiwtz[ ]</td>
<td>Floating Convert to Integer Word with Round to Zero</td>
<td>XL</td>
<td>63</td>
<td>15</td>
</tr>
<tr>
<td>fd[ ]</td>
<td>Floating Divide</td>
<td>A</td>
<td>63</td>
<td>18</td>
</tr>
<tr>
<td>fdiv[ ]</td>
<td>Floating Divide</td>
<td>A</td>
<td>63</td>
<td>18</td>
</tr>
<tr>
<td>fdivs[ ]</td>
<td>Floating Divide Single</td>
<td>A</td>
<td>59</td>
<td>18</td>
</tr>
<tr>
<td>fm[ ]</td>
<td>Floating Multiply</td>
<td>A</td>
<td>63</td>
<td>25</td>
</tr>
<tr>
<td>fma[ ]</td>
<td>Floating Multiply-Add</td>
<td>A</td>
<td>63</td>
<td>29</td>
</tr>
<tr>
<td>fmadd[ ]</td>
<td>Floating Multiply-Add</td>
<td>A</td>
<td>63</td>
<td>29</td>
</tr>
<tr>
<td>fmadds[ ]</td>
<td>Floating Multiply-Add Single</td>
<td>A</td>
<td>59</td>
<td>29</td>
</tr>
<tr>
<td>fmrf[ ]</td>
<td>Floating Move Register</td>
<td>X</td>
<td>63</td>
<td>72</td>
</tr>
<tr>
<td>fms[ ]</td>
<td>Floating Multiply-Subtract</td>
<td>A</td>
<td>63</td>
<td>28</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Class</td>
<td>Address</td>
<td>Size</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------</td>
<td>-------</td>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>fmsub[]</td>
<td>Floating Multiply-Subtract</td>
<td>A</td>
<td>63</td>
<td>28</td>
</tr>
<tr>
<td>fmsubs[]</td>
<td>Floating Multiply-Subtract Single</td>
<td>A</td>
<td>59</td>
<td>28</td>
</tr>
<tr>
<td>fmul[]</td>
<td>Floating Multiply</td>
<td>A</td>
<td>63</td>
<td>25</td>
</tr>
<tr>
<td>fmuls[]</td>
<td>Floating Multiply Single</td>
<td>A</td>
<td>59</td>
<td>25</td>
</tr>
<tr>
<td>fnabs[]</td>
<td>Floating Negative Absolute Value</td>
<td>X</td>
<td>63</td>
<td>136</td>
</tr>
<tr>
<td>fneg[]</td>
<td>Floating Negate</td>
<td>X</td>
<td>63</td>
<td>40</td>
</tr>
<tr>
<td>fnma[]</td>
<td>Floating Negative Multiply-Add</td>
<td>A</td>
<td>63</td>
<td>31</td>
</tr>
<tr>
<td>fnmadd[]</td>
<td>Floating Negative Multiply-Add</td>
<td>A</td>
<td>63</td>
<td>31</td>
</tr>
<tr>
<td>fnmadds[]</td>
<td>Floating Negative Multiply-Add Single</td>
<td>A</td>
<td>59</td>
<td>31</td>
</tr>
<tr>
<td>fnms[]</td>
<td>Floating Negative Multiply-Subtract</td>
<td>A</td>
<td>63</td>
<td>30</td>
</tr>
<tr>
<td>fnmsub[]</td>
<td>Floating Negative Multiply-Subtract</td>
<td>A</td>
<td>63</td>
<td>30</td>
</tr>
<tr>
<td>fnmsubs[]</td>
<td>Floating Negative Multiply-Subtract Single</td>
<td>A</td>
<td>59</td>
<td>30</td>
</tr>
<tr>
<td>frsp[]</td>
<td>Floating Round to Single Precision</td>
<td>X</td>
<td>63</td>
<td>12</td>
</tr>
<tr>
<td>fs[]</td>
<td>Floating Subtract</td>
<td>A</td>
<td>63</td>
<td>20</td>
</tr>
<tr>
<td>fsub[]</td>
<td>Floating Subtract</td>
<td>A</td>
<td>63</td>
<td>20</td>
</tr>
<tr>
<td>fsubs[]</td>
<td>Floating Subtract Single</td>
<td>A</td>
<td>59</td>
<td>20</td>
</tr>
<tr>
<td>icbi</td>
<td>Instruction Cache Block Invalidate</td>
<td>X</td>
<td>31</td>
<td>982</td>
</tr>
<tr>
<td>ics</td>
<td>Instruction Cache Synchronize</td>
<td>X</td>
<td>19</td>
<td>150</td>
</tr>
<tr>
<td>isync</td>
<td>Instruction Synchronize</td>
<td>X</td>
<td>19</td>
<td>150</td>
</tr>
<tr>
<td>l</td>
<td>Load</td>
<td>D</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>lbx</td>
<td>Load Byte-Reversed Indexed</td>
<td>X</td>
<td>31</td>
<td>534</td>
</tr>
<tr>
<td>lbz</td>
<td>Load Byte and Zero</td>
<td>D</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>lbzu</td>
<td>Load Byte and Zero with Update</td>
<td>D</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>lbzux</td>
<td>Load Byte and Zero with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>119</td>
</tr>
<tr>
<td>lbzx</td>
<td>Load Byte and Zero Indexed</td>
<td>X</td>
<td>31</td>
<td>87</td>
</tr>
<tr>
<td>lfd</td>
<td>Load Floating-Point Double</td>
<td>D</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Type</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------</td>
<td>------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>ldw</td>
<td>Load Floating-Point Double with Update</td>
<td>D</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>ldwux</td>
<td>Load Floating-Point Double with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>631</td>
</tr>
<tr>
<td>ldw</td>
<td>Load Floating-Point Double Indexed</td>
<td>X</td>
<td>31</td>
<td>599</td>
</tr>
<tr>
<td>ls</td>
<td>Load Floating-Point Single</td>
<td>D</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>lsu</td>
<td>Load Floating-Point Single with Update</td>
<td>D</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>lsux</td>
<td>Load Floating-Point Single with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>567</td>
</tr>
<tr>
<td>lsx</td>
<td>Load Floating-Point Single Indexed</td>
<td>X</td>
<td>31</td>
<td>535</td>
</tr>
<tr>
<td>lha</td>
<td>Load Half Algebraic</td>
<td>D</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>lhau</td>
<td>Load Half Algebraic with Update</td>
<td>D</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>lhaux</td>
<td>Load Half Algebraic with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>375</td>
</tr>
<tr>
<td>lhax</td>
<td>Load Half Algebraic Indexed</td>
<td>X</td>
<td>31</td>
<td>343</td>
</tr>
<tr>
<td>lhbrx</td>
<td>Load Half Byte-Reversed Indexed</td>
<td>X</td>
<td>31</td>
<td>790</td>
</tr>
<tr>
<td>lhz</td>
<td>Load Half and Zero</td>
<td>D</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>lhzu</td>
<td>Load Half and Zero with Update</td>
<td>D</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>lhzux</td>
<td>Load Half and Zero with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>331</td>
</tr>
<tr>
<td>lhzx</td>
<td>Load Half and Zero Indexed</td>
<td>X</td>
<td>31</td>
<td>279</td>
</tr>
<tr>
<td>lm</td>
<td>Load Multiple</td>
<td>D</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>lmw</td>
<td>Load Multiple Word</td>
<td>D</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>ls CBX</td>
<td>Load String and Compare Byte Indexed</td>
<td>X</td>
<td>31</td>
<td>277</td>
</tr>
<tr>
<td>isi</td>
<td>Load String Immediate</td>
<td>X</td>
<td>31</td>
<td>597</td>
</tr>
<tr>
<td>iswi</td>
<td>Load String Word Immediate</td>
<td>X</td>
<td>31</td>
<td>597</td>
</tr>
<tr>
<td>iswx</td>
<td>Load String Word Indexed</td>
<td>X</td>
<td>31</td>
<td>533</td>
</tr>
<tr>
<td>isx</td>
<td>Load String Indexed</td>
<td>X</td>
<td>31</td>
<td>533</td>
</tr>
<tr>
<td>lu</td>
<td>Load with Update</td>
<td>D</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>lux</td>
<td>Load with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>55</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Mode</td>
<td>Size</td>
<td>Opcode</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------</td>
<td>------</td>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td>lwarx</td>
<td>Load Word and Reserve Indexed</td>
<td>X</td>
<td>31</td>
<td>20</td>
</tr>
<tr>
<td>lwbrx</td>
<td>Load Word Byte-Reversed Indexed</td>
<td>X</td>
<td>31</td>
<td>534</td>
</tr>
<tr>
<td>lwz</td>
<td>Load Word and Zero</td>
<td>D</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>lwzu</td>
<td>Load Word with Zero Update</td>
<td>D</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>lwzux</td>
<td>Load Word and Zero with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>55</td>
</tr>
<tr>
<td>lwzx</td>
<td>Load Word and Zero Indexed</td>
<td>X</td>
<td>31</td>
<td>23</td>
</tr>
<tr>
<td>lx</td>
<td>Load Indexed</td>
<td>X</td>
<td>31</td>
<td>23</td>
</tr>
<tr>
<td>maskg[]</td>
<td>Mask Generate</td>
<td>X</td>
<td>31</td>
<td>29</td>
</tr>
<tr>
<td>maskir[]</td>
<td>Mask Insert from Register</td>
<td>X</td>
<td>31</td>
<td>541</td>
</tr>
<tr>
<td>mcrf</td>
<td>Move Condition Register Field</td>
<td>XL</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>mcrfs</td>
<td>Move to Condition Register from FPSCR</td>
<td>X</td>
<td>63</td>
<td>64</td>
</tr>
<tr>
<td>mcrxr</td>
<td>Move to Condition Register from XER</td>
<td>X</td>
<td>31</td>
<td>512</td>
</tr>
<tr>
<td>mfcr</td>
<td>Move from Condition Register</td>
<td>X</td>
<td>31</td>
<td>19</td>
</tr>
<tr>
<td>mfcs[]</td>
<td>Move from FPSCR</td>
<td>X</td>
<td>63</td>
<td>583</td>
</tr>
<tr>
<td>mfmsr</td>
<td>Move from Machine State Register</td>
<td>X</td>
<td>31</td>
<td>83</td>
</tr>
<tr>
<td>mfspri</td>
<td>Move from Special-Purpose Register</td>
<td>X</td>
<td>31</td>
<td>339</td>
</tr>
<tr>
<td>mfspri</td>
<td>Move from Special-Purpose Register</td>
<td>X</td>
<td>31</td>
<td>595</td>
</tr>
<tr>
<td>mfspri</td>
<td>Move from Segment Register</td>
<td>X</td>
<td>31</td>
<td>659</td>
</tr>
<tr>
<td>mtcfg[]</td>
<td>Move to Condition Register Indirect</td>
<td>X</td>
<td>31</td>
<td>144</td>
</tr>
<tr>
<td>mtfsb0[]</td>
<td>Move to FPSCR Bit 0</td>
<td>X</td>
<td>63</td>
<td>70</td>
</tr>
<tr>
<td>mtfsb1[]</td>
<td>Move to FPSCR Bit 1</td>
<td>X</td>
<td>63</td>
<td>38</td>
</tr>
<tr>
<td>mtfsf[]</td>
<td>Move to FPSCR Fields</td>
<td>XFL</td>
<td>63</td>
<td>711</td>
</tr>
<tr>
<td>mtfsfi[]</td>
<td>Move to FPSCR Field Immediate</td>
<td>X</td>
<td>63</td>
<td>134</td>
</tr>
<tr>
<td>mtemsrr</td>
<td>Move to Machine State Register</td>
<td>X</td>
<td>31</td>
<td>146</td>
</tr>
<tr>
<td>mtpsrr</td>
<td>Move to Special-Purpose Register</td>
<td>X</td>
<td>31</td>
<td>467</td>
</tr>
<tr>
<td>mtsr</td>
<td>Move to Segment Register</td>
<td>X</td>
<td>31</td>
<td>210</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Mode</td>
<td>Opcode</td>
<td>Length</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------------------------------</td>
<td>------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>mtsri</td>
<td>Move to Segment Register Indirect</td>
<td>X</td>
<td>31</td>
<td>242</td>
</tr>
<tr>
<td>mtsrin</td>
<td>Move to Segment Register Indirect</td>
<td>X</td>
<td>31</td>
<td>242</td>
</tr>
<tr>
<td>mul[O][.]</td>
<td>Multiply</td>
<td>XO</td>
<td>31</td>
<td>107</td>
</tr>
<tr>
<td>mulh[O][.]</td>
<td>Multiply High Word</td>
<td>XO</td>
<td>31</td>
<td>75</td>
</tr>
<tr>
<td>mulhw[O][.]</td>
<td>Multiply High Word Unsigned</td>
<td>XO</td>
<td>31</td>
<td>11</td>
</tr>
<tr>
<td>muli</td>
<td>Multiply Immediate</td>
<td>D</td>
<td>07</td>
<td></td>
</tr>
<tr>
<td>mulli</td>
<td>Multiply Low Immediate</td>
<td>D</td>
<td>07</td>
<td></td>
</tr>
<tr>
<td>mulw[O][.]</td>
<td>Multiply Low Word</td>
<td>XO</td>
<td>31</td>
<td>235</td>
</tr>
<tr>
<td>mult[O][.]</td>
<td>Multiply Short</td>
<td>XO</td>
<td>31</td>
<td>235</td>
</tr>
<tr>
<td>nabs[O][.]</td>
<td>Negative Absolute</td>
<td>XO</td>
<td>31</td>
<td>488</td>
</tr>
<tr>
<td>NAND[O][.]</td>
<td>NAND</td>
<td>X</td>
<td>31</td>
<td>476</td>
</tr>
<tr>
<td>neg[O][.]</td>
<td>Negate</td>
<td>XO</td>
<td>31</td>
<td>104</td>
</tr>
<tr>
<td>nor[O][.]</td>
<td>NOR</td>
<td>X</td>
<td>31</td>
<td>124</td>
</tr>
<tr>
<td>or[O][.]</td>
<td>OR</td>
<td>X</td>
<td>31</td>
<td>444</td>
</tr>
<tr>
<td>orc[O][.]</td>
<td>OR with Complement</td>
<td>X</td>
<td>31</td>
<td>412</td>
</tr>
<tr>
<td>ori</td>
<td>OR Immediate</td>
<td>D</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>oril</td>
<td>OR Immediate Lower</td>
<td>D</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>oris</td>
<td>OR Immediate Shifted</td>
<td>D</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>oriu</td>
<td>OR Immediate Upper</td>
<td>D</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>rfi</td>
<td>Return from Interrupt</td>
<td>X</td>
<td>19</td>
<td>50</td>
</tr>
<tr>
<td>rlim[O][.]</td>
<td>Rotate Left Immediate then Mask Insert</td>
<td>M</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>rlinm[O][.]</td>
<td>Rotate Left Immediate then AND with Mask</td>
<td>M</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>rlim[O][.]</td>
<td>Rotate Left then Mask Insert</td>
<td>M</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>rlinn[O][.]</td>
<td>Rotate Left then AND with Mask</td>
<td>M</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>rliwimi[O][.]</td>
<td>Rotate Left Word Immediate then Mask Insert</td>
<td>M</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>rliwinm[O][.]</td>
<td>Rotate Left Word Immediate then AND with Mask</td>
<td>M</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>rliwnm[O][.]</td>
<td>Rotate Left Word then AND with Mask</td>
<td>M</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>rrib[O][.]</td>
<td>Rotate Right and Insert Bit</td>
<td>X</td>
<td>31</td>
<td>537</td>
</tr>
<tr>
<td>sc</td>
<td>System Call</td>
<td>SC</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>sfo[O][.]</td>
<td>Subtract from</td>
<td>XO</td>
<td>31</td>
<td>08</td>
</tr>
<tr>
<td>sfeo[O][.]</td>
<td>Subtract from Extended</td>
<td>XO</td>
<td>31</td>
<td>136</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>sfi</strong></td>
<td>Subtract from Immediate</td>
<td>D</td>
<td>08</td>
<td></td>
</tr>
<tr>
<td><strong>sfme</strong>[D]</td>
<td>Subtract from Minus One Extended</td>
<td>XO</td>
<td>31</td>
<td>232</td>
</tr>
<tr>
<td><strong>sfze</strong>[D]</td>
<td>Subtract from Zero Extended</td>
<td>XO</td>
<td>31</td>
<td>200</td>
</tr>
<tr>
<td><strong>si</strong></td>
<td>Subtract Immediate</td>
<td>D</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td><strong>si</strong></td>
<td>Subtract Immediate and Record</td>
<td>D</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td><strong>sl[.]</strong></td>
<td>Shift Left</td>
<td>X</td>
<td>31</td>
<td>24</td>
</tr>
<tr>
<td><strong>sle[.]</strong></td>
<td>Shift Left Extended</td>
<td>X</td>
<td>31</td>
<td>153</td>
</tr>
<tr>
<td><strong>sleq[.]</strong></td>
<td>Shift Left Extended with MQ</td>
<td>X</td>
<td>31</td>
<td>217</td>
</tr>
<tr>
<td><strong>sliq[.]</strong></td>
<td>Shift Left Immediate with MQ</td>
<td>X</td>
<td>31</td>
<td>184</td>
</tr>
<tr>
<td><strong>slliq[.]</strong></td>
<td>Shift Left Long Immediate with MQ</td>
<td>X</td>
<td>31</td>
<td>248</td>
</tr>
<tr>
<td><strong>slq[.]</strong></td>
<td>Shift Left with MQ</td>
<td>X</td>
<td>31</td>
<td>152</td>
</tr>
<tr>
<td><strong>slw[.]</strong></td>
<td>Shift Left Word</td>
<td>X</td>
<td>31</td>
<td>24</td>
</tr>
<tr>
<td><strong>sr[.]</strong></td>
<td>Shift Right</td>
<td>X</td>
<td>31</td>
<td>536</td>
</tr>
<tr>
<td><strong>sra[.]</strong></td>
<td>Shift Right Algebraic</td>
<td>X</td>
<td>31</td>
<td>792</td>
</tr>
<tr>
<td><strong>sral[.]</strong></td>
<td>Shift Right Algebraic Immediate</td>
<td>X</td>
<td>31</td>
<td>824</td>
</tr>
<tr>
<td><strong>sraiq[.]</strong></td>
<td>Shift Right Algebraic Immediate with MQ</td>
<td>X</td>
<td>31</td>
<td>952</td>
</tr>
<tr>
<td><strong>sraq[.]</strong></td>
<td>Shift Right Algebraic with MQ</td>
<td>X</td>
<td>31</td>
<td>920</td>
</tr>
<tr>
<td><strong>sraw[.]</strong></td>
<td>Shift Right Algebraic Word</td>
<td>X</td>
<td>31</td>
<td>792</td>
</tr>
<tr>
<td><strong>srawi[.]</strong></td>
<td>Shift Right Algebraic Word Immediate</td>
<td>X</td>
<td>31</td>
<td>824</td>
</tr>
<tr>
<td><strong>sre[.]</strong></td>
<td>Shift Right Extended</td>
<td>X</td>
<td>31</td>
<td>665</td>
</tr>
<tr>
<td><strong>srea[.]</strong></td>
<td>Shift Right Extended Algebraic</td>
<td>X</td>
<td>31</td>
<td>921</td>
</tr>
<tr>
<td><strong>sreq[.]</strong></td>
<td>Shift Right Extended with MQ</td>
<td>X</td>
<td>31</td>
<td>729</td>
</tr>
<tr>
<td><strong>sriq[.]</strong></td>
<td>Shift Right Immediate with MQ</td>
<td>X</td>
<td>31</td>
<td>696</td>
</tr>
<tr>
<td><strong>srlq[.]</strong></td>
<td>Shift Right Long Immediate with MQ</td>
<td>X</td>
<td>31</td>
<td>760</td>
</tr>
<tr>
<td><strong>srlaq[.]</strong></td>
<td>Shift Right Long with MQ</td>
<td>X</td>
<td>31</td>
<td>728</td>
</tr>
<tr>
<td><strong>srq[.]</strong></td>
<td>Shift Right with MQ</td>
<td>X</td>
<td>31</td>
<td>664</td>
</tr>
<tr>
<td><strong>srw[.]</strong></td>
<td>Shift Right Word</td>
<td>X</td>
<td>31</td>
<td>536</td>
</tr>
<tr>
<td><strong>st</strong></td>
<td>Store</td>
<td>D</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Mod</td>
<td>Rs</td>
<td>Opcode</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-----</td>
<td>----</td>
<td>--------</td>
</tr>
<tr>
<td>stb</td>
<td>Store Byte</td>
<td>D</td>
<td></td>
<td>38</td>
</tr>
<tr>
<td>stbrx</td>
<td>Store Byte-Reversed Indexed</td>
<td>X</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>stbu</td>
<td>Store Byte with Update</td>
<td>D</td>
<td></td>
<td>39</td>
</tr>
<tr>
<td>stbux</td>
<td>Store Byte with Update Indexed</td>
<td>X</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>stbx</td>
<td>Store Byte Indexed</td>
<td>X</td>
<td></td>
<td>215</td>
</tr>
<tr>
<td>stfd</td>
<td>Store Floating-Point Double</td>
<td>D</td>
<td></td>
<td>54</td>
</tr>
<tr>
<td>stfdx</td>
<td>Store Floating-Point Double Indexed</td>
<td>X</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>stfx</td>
<td>Store Floating-Point Double Indexed</td>
<td>X</td>
<td></td>
<td>727</td>
</tr>
<tr>
<td>stfs</td>
<td>Store Floating-Point Single</td>
<td>D</td>
<td></td>
<td>52</td>
</tr>
<tr>
<td>stfsu</td>
<td>Store Floating-Point Single with Update</td>
<td>D</td>
<td></td>
<td>53</td>
</tr>
<tr>
<td>stfsux</td>
<td>Store Floating-Point Single with Update Indexed</td>
<td>X</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>stfsx</td>
<td>Store Floating-Point Single Indexed</td>
<td>X</td>
<td></td>
<td>663</td>
</tr>
<tr>
<td>sth</td>
<td>Store Half</td>
<td>D</td>
<td></td>
<td>44</td>
</tr>
<tr>
<td>sthbrx</td>
<td>Store Half Byte-Reversed Indexed</td>
<td>X</td>
<td></td>
<td>918</td>
</tr>
<tr>
<td>sthu</td>
<td>Store Half with Update</td>
<td>D</td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>sthux</td>
<td>Store Half with Update Indexed</td>
<td>X</td>
<td></td>
<td>439</td>
</tr>
<tr>
<td>sthx</td>
<td>Store Half Indexed</td>
<td>X</td>
<td></td>
<td>407</td>
</tr>
<tr>
<td>stm</td>
<td>Store Multiple</td>
<td>D</td>
<td></td>
<td>47</td>
</tr>
<tr>
<td>stmw</td>
<td>Store Multiple Word</td>
<td>D</td>
<td></td>
<td>47</td>
</tr>
<tr>
<td>stsi</td>
<td>Store String Immediate</td>
<td>X</td>
<td></td>
<td>725</td>
</tr>
<tr>
<td>stswi</td>
<td>Store String Word Immediate</td>
<td>X</td>
<td></td>
<td>725</td>
</tr>
<tr>
<td>stswx</td>
<td>Store String Word Indexed</td>
<td>X</td>
<td></td>
<td>661</td>
</tr>
<tr>
<td>stsx</td>
<td>Store String Indexed</td>
<td>X</td>
<td></td>
<td>661</td>
</tr>
<tr>
<td>stu</td>
<td>Store with Update</td>
<td>D</td>
<td></td>
<td>37</td>
</tr>
<tr>
<td>stux</td>
<td>Store with Update Indexed</td>
<td>X</td>
<td></td>
<td>183</td>
</tr>
<tr>
<td>stw</td>
<td>Store</td>
<td>D</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Type</td>
<td>Offset</td>
<td>Code</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>stwbrx</td>
<td>Store Word Byte-Reversed Indexed</td>
<td>X</td>
<td>31</td>
<td>662</td>
</tr>
<tr>
<td>stwcx</td>
<td>Store Word Conditional Indexed</td>
<td>X</td>
<td>31</td>
<td>150</td>
</tr>
<tr>
<td>stwu</td>
<td>Store Word with Update</td>
<td>D</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>stwux</td>
<td>Store Word with Update Indexed</td>
<td>X</td>
<td>31</td>
<td>183</td>
</tr>
<tr>
<td>stwx</td>
<td>Store Word Indexed</td>
<td>X</td>
<td>31</td>
<td>151</td>
</tr>
<tr>
<td>stx</td>
<td>Store Indexed</td>
<td>X</td>
<td>31</td>
<td>151</td>
</tr>
<tr>
<td>subf[o][]</td>
<td>Subtract from</td>
<td>XO</td>
<td>31</td>
<td>40</td>
</tr>
<tr>
<td>subfc[o][]</td>
<td>Subtract from Carrying</td>
<td>XO</td>
<td>31</td>
<td>08</td>
</tr>
<tr>
<td>subfe[o][]</td>
<td>Subtract from Extended</td>
<td>XO</td>
<td>31</td>
<td>136</td>
</tr>
<tr>
<td>subfic</td>
<td>Subtract from Immediate Carrying</td>
<td>D</td>
<td>08</td>
<td></td>
</tr>
<tr>
<td>subfme[o][]</td>
<td>Subtract from Minus One Extended</td>
<td>XO</td>
<td>31</td>
<td>232</td>
</tr>
<tr>
<td>subfze[o][]</td>
<td>Subtract from Zero Extended</td>
<td>XO</td>
<td>31</td>
<td>200</td>
</tr>
<tr>
<td>sync</td>
<td>Synchronize</td>
<td>X</td>
<td>31</td>
<td>598</td>
</tr>
<tr>
<td>t</td>
<td>Trap</td>
<td>X</td>
<td>31</td>
<td>04</td>
</tr>
<tr>
<td>ti</td>
<td>Trap Immediate</td>
<td>D</td>
<td>03</td>
<td></td>
</tr>
<tr>
<td>tlbie</td>
<td>Translation Look-aside Buffer Invalidate Entry</td>
<td>X</td>
<td>31</td>
<td>306</td>
</tr>
<tr>
<td>tw</td>
<td>Trap Word</td>
<td>X</td>
<td>31</td>
<td>04</td>
</tr>
<tr>
<td>twi</td>
<td>Trap Word Immediate</td>
<td>D</td>
<td>03</td>
<td></td>
</tr>
<tr>
<td>xor[.]</td>
<td>XOR</td>
<td>X</td>
<td>31</td>
<td>316</td>
</tr>
<tr>
<td>xori</td>
<td>XOR Immediate</td>
<td>D</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>xoril</td>
<td>XOR Immediate Lower</td>
<td>D</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>xoris</td>
<td>XOR Immediate Shift</td>
<td>D</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>xoriu</td>
<td>XOR Immediate Upper</td>
<td>D</td>
<td>27</td>
<td></td>
</tr>
</tbody>
</table>
Appendix H. Value Definitions

Bits 0-5
These bits represent the opcode portion of the machine instruction.

Bits 6-30
These bits contain fields defined according to the values below. Note that many instructions also contain extended opcodes, which occupy some portion of the bits in this range. Refer to specific instructions to understand the format utilized.

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>/, //, ///</td>
<td>Reserved/unused; nominally zero (0).</td>
</tr>
<tr>
<td>A</td>
<td>Pseudonym for RA in some diagrams.</td>
</tr>
</tbody>
</table>
| AA | Absolute address bit.  
  • 0 - The immediate field represents an address relative to the current instruction address.  
  • 1 - The immediate field represents an absolute address. |
<p>| B | Pseudonym for RB in some diagrams. |
| BA | Specifies source condition register bit for operation. |
| BB | Specifies source condition register bit for operation. |
| BD | Specifies a 14-bit value used as the branch displacement. |
| BF | Specifies condition register field 0-7 which indicates the result of a compare. |
| BFA | Specifies source condition register field for operation. |
| BI | Specifies bit in condition register for condition comparison. |
| BO | Specifies branch option field used in instruction. |
| BT | Specifies target condition register bit where result of operation is stored. |
| D | Specifies 16-bit two’s-complement integer sign extended to 32 bits. |
| DS | Specifies a 14-bit field used as an immediate value for the calculation of an effective address (EA). |
| FL1 | Specifies field for optional data passing the SVC routine. |
| FL2 | Specifies field for optional data passing the SVC routine. |
| FLM | Specifies field mask. |
| FRA | Specifies source floating-point register for operation. |
| FRB | Specifies source floating-point register for operation. |
| FRC | Specifies source floating-point register for operation. |
| FRS | Specifies source floating-point register of stored data. |
| FRT | Specifies target floating-point register for operation. |
| FXM | Specifies field mask. |
| I | Specifies source immediate value for operation. |
| L | Must be set to 0 for the 32-bit subset architecture. |
| LEV | Specifies the execution address. |
| LI | Immediate field specifying a 24-bit signed two’s complement integer that is concatenated on the right with 0b00 and sign-extended to 64 bits (32 bits in 32-bit implementations). |
| LK | If LK=1, the effective address of the instruction following the branch instruction is place into the link register. |</p>
<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB</td>
<td>Specifies the begin value (bit number) of the mask for the operation.</td>
</tr>
<tr>
<td>ME</td>
<td>Specifies the end value (bit number) of the mask for the operation.</td>
</tr>
<tr>
<td>NB</td>
<td>Specifies the byte count for the operation.</td>
</tr>
<tr>
<td>OE</td>
<td>Specifies that the overflow bits in the Fixed-Point Exception register are affected if the operation results in overflow.</td>
</tr>
<tr>
<td>RA</td>
<td>Specifies the source general-purpose register for the operation.</td>
</tr>
<tr>
<td>RB</td>
<td>Specifies the source general-purpose register for the operation.</td>
</tr>
<tr>
<td>RS</td>
<td>Specifies the source general-purpose register for the operation.</td>
</tr>
<tr>
<td>RT</td>
<td>Specifies the target general-purpose register where the operation is stored.</td>
</tr>
<tr>
<td>S</td>
<td>Pseudonym for RS in some diagrams.</td>
</tr>
<tr>
<td>SA</td>
<td>Documented in the <code>svc</code> instruction.</td>
</tr>
<tr>
<td>SH</td>
<td>Specifies the (immediate) shift value for the operation.</td>
</tr>
<tr>
<td>SI</td>
<td>Specifies the 16-bit signed integer for the operation.</td>
</tr>
<tr>
<td>SIMM</td>
<td>16-bit two's-complement value which will be sign-extended for comparison.</td>
</tr>
<tr>
<td>SPR</td>
<td>Specifies the source special purpose register for the operation.</td>
</tr>
<tr>
<td>SR</td>
<td>Specifies the source segment register for the operation.</td>
</tr>
<tr>
<td>ST</td>
<td>Specifies the target segment register for the operation.</td>
</tr>
<tr>
<td>TO</td>
<td>Specifies TO bits that are ANDed with compare results.</td>
</tr>
<tr>
<td>U</td>
<td>Specifies source immediate value for operation.</td>
</tr>
<tr>
<td>UI</td>
<td>Specifies 16-bit unsigned integer for operation.</td>
</tr>
</tbody>
</table>

### Bit 31

Bit 31 is the record bit.

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Does not update the condition register.</td>
</tr>
<tr>
<td>1</td>
<td>Updates the condition register to reflect the result of the operation.</td>
</tr>
</tbody>
</table>
Appendix I. Vector Processor

This appendix provides an overview of the vector processor, as well as AIX ABI extensions and linkage conventions in support of the vector processor.

For more information on the vector processor and vector processor instructions, see the AltiVec Technology Programming Environments Manual.

Storage Operands and Alignment

All vector data types are 16 bytes in size, and must be aligned on a 16-byte (quadword) boundary. Aggregates containing vector types must follow normal conventions of aligning the aggregate to the requirement of its largest member. If an aggregate containing a vector type is packed, then there is no guarantee of 16-byte alignment of the vector type.

Table 38. Data Types

<table>
<thead>
<tr>
<th>Contents</th>
<th>New C/C++ Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 unsigned char</td>
<td>vector unsigned char</td>
</tr>
<tr>
<td>16 signed char</td>
<td>vector signed char</td>
</tr>
<tr>
<td>16 unsigned char</td>
<td>vector bool char</td>
</tr>
<tr>
<td>8 unsigned short</td>
<td>vector unsigned short</td>
</tr>
<tr>
<td>8 signed short</td>
<td>vector signed short</td>
</tr>
<tr>
<td>8 unsigned short</td>
<td>vector bool short</td>
</tr>
<tr>
<td>4 unsigned int</td>
<td>vector unsigned int</td>
</tr>
<tr>
<td>4 signed int</td>
<td>vector signed int</td>
</tr>
<tr>
<td>4 unsigned int</td>
<td>vector bool int</td>
</tr>
<tr>
<td>4 float</td>
<td>vector float</td>
</tr>
</tbody>
</table>

Register Usage Conventions

The PowerPC Vector Extension architecture adds 32 vector registers (VRs). Each VR is 128 bits wide. There is also a 32-bit special purpose register (VRSAVE), and a 32-bit vector status and control register (VSCR). The VR conventions table shows how VRs are used:

Table 39. VR Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Status</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0</td>
<td>Volatile</td>
<td>Scratch register.</td>
</tr>
<tr>
<td>VR1</td>
<td>Volatile</td>
<td>Scratch register.</td>
</tr>
<tr>
<td>VR2</td>
<td>Volatile</td>
<td>First vector argument. First vector of function return value.</td>
</tr>
<tr>
<td>VR3</td>
<td>Volatile</td>
<td>Second vector argument, scratch.</td>
</tr>
<tr>
<td>VR4</td>
<td>Volatile</td>
<td>Third vector argument, scratch.</td>
</tr>
<tr>
<td>VR5</td>
<td>Volatile</td>
<td>Fourth vector argument, scratch.</td>
</tr>
<tr>
<td>VR6</td>
<td>Volatile</td>
<td>Fifth vector argument, scratch.</td>
</tr>
<tr>
<td>VR7</td>
<td>Volatile</td>
<td>Sixth vector argument, scratch.</td>
</tr>
<tr>
<td>VR8</td>
<td>Volatile</td>
<td>Seventh vector argument, scratch.</td>
</tr>
<tr>
<td>VR9</td>
<td>Volatile</td>
<td>Eighth vector argument, scratch.</td>
</tr>
</tbody>
</table>
Table 39. VR Conventions (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Status</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR10</td>
<td>Volatile</td>
<td>Ninth vector argument, scratch.</td>
</tr>
<tr>
<td>VR11</td>
<td>Volatile</td>
<td>Tenth vector argument, scratch.</td>
</tr>
<tr>
<td>VR12</td>
<td>Volatile</td>
<td>Eleventh vector argument, scratch.</td>
</tr>
<tr>
<td>VR13</td>
<td>Volatile</td>
<td>Twelfth vector argument, scratch.</td>
</tr>
<tr>
<td>VR14:19</td>
<td>Volatile</td>
<td>Scratch.</td>
</tr>
<tr>
<td>VR20:31</td>
<td>Reserved (default mode) Non-Volatile (extended ABI mode)</td>
<td>When the default vector enabled mode is used, these registers are reserved, and must not be used. In the extended ABI Vector enabled mode, these registers are non-volatile and their values are preserved across function calls.</td>
</tr>
<tr>
<td>VRSAVE</td>
<td>Reserved</td>
<td>In the AIX ABI, VRSAVE is not used. An ABI-compliant program must not use or alter VRSAVE.</td>
</tr>
<tr>
<td>VSCR</td>
<td>Volatile</td>
<td>Vector status and control register. Contains saturation status bit and non-Java mode control bit.</td>
</tr>
</tbody>
</table>

The AltiVec Programming Interface Specification defines the VRSAVE register to be used as a bitmask of vector registers in use. AIX requires that an application never modify the VRSAVE register.

Runtime Stack

The runtime stack begins quadword aligned for both 32-bit and 64-bit processes. The conventions that are discussed in the four following paragraphs are defined for stack save areas for VRs, as well as conventions for vector parameters passed on the stack.

VRSAVE is not recognized by the AIX ABI, and should not be used or altered by ABI-compliant programs. The VRSAVE runtime stack save location remains reserved for compatibility with legacy compiler linkage convention.

The alignment padding space will be either 0, 4, 8, or 12 bytes as necessary to align the vector save area to a quadword boundary. Before use, any non-volatile VR must be saved in its VR save area on the stack, beginning with VR31, continuing down to VR20. Local variables of vector data type that need to be saved to memory are saved to the same stack frame region used for local variables of other types, but on a 16-byte boundary.

The stack floor remains at 220 bytes for 32-bit mode and 288 bytes for 64-bit mode. In the event that a function needs to save non-volatile general purpose registers (GPRs), floating-point registers (FPRs), and VRs totaling more than the respective mode’s floor size, the function must first atomically update the stack pointer prior to saving the non-volatile VRs.

Any vector variables within the local variable region must be aligned to a 16-byte boundary.

The 32-bit runtime stack looks like the following (pre-prolog):

Table 40. Example of a 32-bit Runtime Stack

<table>
<thead>
<tr>
<th>Sp -&gt;</th>
<th>Back chain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FPR31 (if needed)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>-nFPRs*8</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>GPR31 (if needed)</td>
</tr>
</tbody>
</table>
Table 40. Example of a 32-bit Runtime Stack (continued)

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-nGPRs*4</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>VRSAVE</td>
</tr>
<tr>
<td></td>
<td>Alignment padding (to 16-byte boundary)</td>
</tr>
<tr>
<td>-nVRs*16</td>
<td>...</td>
</tr>
<tr>
<td>-220(max)</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Local variables</td>
</tr>
<tr>
<td>NF+24</td>
<td>Parameter List Area</td>
</tr>
<tr>
<td>NF+20</td>
<td>Saved TOC</td>
</tr>
<tr>
<td>NF+16</td>
<td>Reserved (binder)</td>
</tr>
<tr>
<td>NF+12</td>
<td>Reserved (compiler)</td>
</tr>
<tr>
<td>NF+8</td>
<td>Saved LR</td>
</tr>
<tr>
<td>NF+4</td>
<td>Saved CR</td>
</tr>
<tr>
<td>NF -&gt;</td>
<td>Sp (after NF-newframe allocated)</td>
</tr>
</tbody>
</table>

The 64-bit runtime stack looks like the following (pre-prolog):

Table 41. Example of a 64-bit Runtime Stack

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sp -&gt;</td>
<td>Back chain</td>
</tr>
<tr>
<td></td>
<td>FPR31 (if needed)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>-nFPRs*8</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>GPR31 (if needed)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>-nGPRs*8</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>VRSAVE</td>
</tr>
<tr>
<td></td>
<td>Alignment padding (to 16-byte boundary)</td>
</tr>
<tr>
<td></td>
<td>VR31 (if needed)</td>
</tr>
<tr>
<td>-nVRs*16</td>
<td>...</td>
</tr>
<tr>
<td>-288(max)</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Local variables</td>
</tr>
<tr>
<td>NF+48</td>
<td>Parameter List Area</td>
</tr>
<tr>
<td>NF+40</td>
<td>Saved TOC</td>
</tr>
<tr>
<td>NF+32</td>
<td>Reserved (binder)</td>
</tr>
<tr>
<td>NF+24</td>
<td>Reserved (compiler)</td>
</tr>
<tr>
<td>NF+16</td>
<td>Saved LR</td>
</tr>
<tr>
<td>NF+8</td>
<td>Saved CR</td>
</tr>
<tr>
<td>NF -&gt;</td>
<td>Sp (after NF-newframe allocated)</td>
</tr>
</tbody>
</table>
Vector Register Save and Restore Procedures

The vector save and restore functions listed below are provided by the system (libc) as an aid to language compilers.

On entry, \( r0 \) must contain the 16-byte aligned address just above the vector save area. \( r0 \) is left unchanged, but \( r12 \) is modified.

```assembly
_savev20:  addi r12, r0, -192
            stvx v20, r12, r0  # save v20

_savev21:  addi r12, r0, -176
            stvx v21, r12, r0  # save v21

_savev22:  addi r12, r0, -160
            stvx v22, r12, r0  # save v22

_savev23:  addi r12, r0, -144
            stvx v23, r12, r0  # save v23

_savev24:  addi r12, r0, -128
            stvx v24, r12, r0  # save v24

_savev25:  addi r12, r0, -112
            stvx v25, r12, r0  # save v25

_savev26:  addi r12, r0, -96
            stvx v26, r12, r0  # save v26

_savev27:  addi r12, r0, -80
            stvx v27, r12, r0  # save v27

_savev28:  addi r12, r0, -64
            stvx v28, r12, r0  # save v28

_savev29:  addi r12, r0, -48
            stvx v29, r12, r0  # save v29

_savev30:  addi r12, r0, -32
            stvx v30, r12, r0  # save v30

_savev31:  addi r12, r0, -16
            stvx v31, r12, r0  # save v31
            br

_restv20:  addi r12, r0, -192
            lvx v20, r12, r0  # restore v20

_restv21:  addi r12, r0, -176
            lvx v21, r12, r0  # restore v21

_restv22:  addi r12, r0, -160
            lvx v22, r12, r0  # restore v22

_restv23:  addi r12, r0, -144
```


Procedure Calling Sequence

The following sections describe the procedure calling conventions with respect to argument passing and return values.

Argument Passing

The first twelve vector parameters to a function are placed in registers VR2 through VR13. Unnecessary vector parameter registers contain undefined values upon entry to the function. Non-variable length argument list vector parameters are not shadowed in GPRs. Any additional vector parameters (13th and beyond) are passed through memory on the program stack, 16-byte aligned, in their appropriate mapped location within the parameter region corresponding to their position in the parameter list.

For variable length argument lists, va_list continues to be a pointer to the memory location of the next parameter. When va_arg() accesses a vector type, va_list must first be aligned to a 16-byte boundary. The receiver and consumer of a variable length argument list is responsible for performing this alignment prior to retrieving the vector type parameter.

A non-packed structure or union passed by value that has a vector member anywhere within it will be aligned to a 16-byte boundary on the stack.

A function that takes a variable length argument list has all parameters mapped in the argument area ordered and aligned according to their type. The first eight words (32-bit) or doublewords (64-bit) of a variable length argument list are shadowed in GPRs r3 through r10. This includes vector parameters. The tables below illustrate variable length argument list parameters:

Table 42. 32-bit Variable Length Argument List Parameters (post-prolog)

<table>
<thead>
<tr>
<th>OldSp -&gt;</th>
<th>Back chain (bc)</th>
</tr>
</thead>
</table>

Appendix I. Vector Processor 601
Table 42. 32-bit Variable Length Argument List Parameters (post-prolog)  (continued)

| -nFPRs*8    | ... |
| -nGPRs*4    | ... |
| -220(max)   | VRSAVE
|             | Local variables |
| SP+56       | ... |
| SP+52       | PW7  | Vector Parm 2b, shadow in GPR10 |
| SP+48       | PW6  | Vector Parm 2a, shadow in GPR9 |
| SP+44       | PW5  | Vector Parm 1d, shadow in GPR8 |
| SP+40       | PW4  | Vector Parm 1c, shadow in GPR7 |
| SP+36       | PW3  | Vector Parm 1b, shadow in GPR6 |
| SP+32       | PW2  | Vector Parm 1a, shadow in GPR5 |
| SP+28       | PW1  |
| SP+24       | PW0  |
| SP+20       | Saved TOC |
| SP+16       | Reserved (binder) |
| SP+12       | Reserved (compiler) |
| SP+8        | Saved LR |
| SP+4        | Saved CR |
| SP ->       | OldSP |

Table 43. 64-bit variable length argument list parameters (post-prolog)

| OldSp -> | Back chain (bc) |
| -nFPRs*8 | ... |
| -nGPRs*8 | ... |
| -288(max) | VRSAVE |
|          | Local variables |
| SP+112   | ... |
| SP+104   | PW7  | Vector Parm 4c, 4d, shadow in GPR10 |
| SP+96    | PW6  | Vector Parm 4a, 4b, shadow in GPR9 |
| SP+88    | PW5  | Vector Parm 3c, 3d, shadow in GPR8 |
| SP+80    | PW4  | Vector Parm 3a, 3b, shadow in GPR7 |
| SP+72    | PW3  | Vector Parm 2c, 2d, shadow in GPR6 |
| SP+64    | PW2  | Vector Parm 2a, 2b, shadow in GPR5 |
| SP+56    | PW1  | Vector Parm 1c, 1d, shadow in GPR4 |
| SP+48    | PW0  | Vector Parm 1a, 1b, shadow in GPR3 |
| SP+40    | Saved TOC |
| SP+32    | Reserved (binder) |
| SP+24    | Reserved (compiler) |
| SP+16    | Saved LR |
| SP+8     | Saved CR |
| SP ->    | OldSP |
**Function Return Values**

Functions that have a return value declared as a vector data type place the return value in VR2. Any function that returns a vector type or has vector parameters requires a function prototype. This avoids the compiler needing to shadow the VRs in GPRs for the general case.

**Traceback Tables**

The traceback table information is extended to provide the information necessary to determine the presence of vector state in the stack frame for a function. One of the unused bits from the `spare3` field is claimed to indicate that the traceback table contains vector information. So the following changes are made to the mandatory traceback table information:

```c
unsigned spare3:1;  /* Spare bit */
unsigned has_vec:1;  /* Set if optional vector info is present */
```

If the `has_vec` field is set, then the optional `parminfo` field is present as well as the following optional extended information. The new optional vector information, if present, would follow the other defined optional fields and would be after the `alloca_reg` optional information.

```c
unsigned vr_saved:6;  /* Number of non-volatile vector registers saved */
                     /* first register saved is assumed to be */
                     /* 32 - vr_saved */
unsigned saves_vrsave:1;  /* Set if vrsave is saved on the stack */
unsigned has_varargs:1;  /* Set if function has a variable length argument list */
unsigned vectorparms:7;  /* Number of vector parameters if not variable */
                     /* argument list. Otherwise the mandatory field */
                     /* parmonstk field must be set */
unsigned vec_present:1;  /* Set if routine performs vector instructions */
unsigned char vecparminfo[4];  /* bitmask array for each vector parm in */
                     /* order as found in the original parminfo, */
                     /* describes the type of vector: */
                     /* b '00' = vector char */
                     /* b '01' = vector short */
                     /* b '10' = vector int */
                     /* b '11' = vector float */
```

If `vectorparms` is non-zero, then the `parminfo` field is interpreted as:

```c
/* b '00' = fixed parameter */
/* b '01' = vector parameter */
/* b '10' = single-precision float parameter */
/* b '11' = double-precision float parameter */
```

**Debug Stabstrings**

New stabstring codes are defined to specify the location of objects in VRs. A code of "X" describes a parameter passed by value in the specified vector register. A code of "x" describes a local variable residing in the specified VR. The existing storage classes of C_LSYM (local variable on stack), C_PSYM (parameter on stack) are used for vector data types in memory where the corresponding stabstrings use arrays of existing fundamental types to represent the data. The existing storage classes of C_RPSYM (parameter in register), and C_RSYM (variable in register) are used in conjunction with the new stabstring codes 'X' and 'x' respectively to represent vector data types in vector registers.
Legacy ABI Compatibility and Interoperability

Due to the nature of interfaces such as `setjmp()`, `longjmp()`, `sigsetjmp()`, `siglongjmp()`, `_setjmp()`, `_longjmp()`, `getcontext()`, `setcontext()`, `makecontext()`, and `swapcontext()`, which must save and restore non-volatile machine state, there is risk introduced when considering dependencies between legacy and vector extended ABI modules. To complicate matters, the `setjmp` family of functions in `libc` reside in a static member of `libc`, which means every existing AIX binary has a statically bound copy of the `setjmp` and others that existed with the version of AIX it was linked against. Furthermore, existing AIX binaries have `jmpbufs` and `ucontext` data structure definitions that are insufficient to house any additional non-volatile vector register state.

Any cases where previous versions of modules and new modules interleave calls, or call-backs, where a previous version of a module could perform a `longjmp()` or `setcontext()` bypassing normal linkage convention of a vector extended module, there is risk of compromising non-volatile VR state.

For this reason, while the AIX ABI defines non-volatile VRs, the default compilation mode when using vectors (AltiVec) in AIX compilers will be to not use any of the non-volatile VRs. This results in a default compilation environment that safely allows exploitation of vectors (AltiVec) while introducing no risk with respect to interoperability with previous-version binaries.

For applications where interoperability and module dependence is completely known, an additional compilation option can be enabled that allows the use of non-volatile VRs. This mode should only be used when all dependent previous-version modules and behaviors are fully known and understood as either having no dependence on functions such as `setjmp()`, `sigsetjmp()`, `_setjmp()`, or `getcontext()`, or ensuring that all module transitions are performed through normal subroutine linkage convention, and that no call-backs to an upstream previous-version module are used.

This approach allows for a completely safe mode of exploitation of vectors (AltiVec), which is the default mode, while also allowing for explicit tuning and further optimization with use of non-volatile registers in cases where the risks are known. It also provides a flexible ABI and architecture for the future.

The default AltiVec compilation environment predefines `__VEC__`, as described in the `AltiVec Programming Interface Manual`.

When the option to use non-volatile VRs is enabled, the compilation environment must also predefine `__EXTABI__`. This should also be defined when you are compiling or recompiling non-vector enabled modules that will interact with vector-enabled modules that are enabled to utilize non-volatile VRs.
Appendix J. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service. IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:
IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you. This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:
IBM Corporation
Dept. LRAS/Bldg. 905
11501 Burnet Road
Austin, TX 78758-3498
U.S.A.
Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.
The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and
cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademark
The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
AIX 5L
IBM
POWER
POWER4
POWER5
POWER Architecture
PowerPC
PowerPC 601
PowerPC 603
PowerPC Architecture
RS/6000

UNIX is a registered trademark of The Open Group in the United States and other countries.

Portions of Chapter 8 are used with the permission of Motorola, Inc.

Other company, product, or service names may be the trademarks or service marks of others.
Index

Special characters

- .align pseudo-op 466
- .bb pseudo-op 467
- .bc pseudo-op 468
- .bf pseudo-op 468
- .bi pseudo-op 469
- .bs pseudo-op 469
- .byte pseudo-op 470
- .comm pseudo-op 471
- .csect pseudo-op 473
- .double pseudo-op 475
- .drop pseudo-op 476
- .dsect pseudo-op 477
- .eb pseudo-op 479
- .ec pseudo-op 479
- .ef pseudo-op 480
- .ei pseudo-op 480
- .es pseudo-op 481
- .extern pseudo-op 481
- .file pseudo-op 482
- .float pseudo-op 483
- .function pseudo-op 483
- .globl pseudo-op 484
- .hash pseudo-op 485
- .lcomm pseudo-op 486
- .lglobl pseudo-op 487
- .line pseudo-op 488
- .llong pseudo-op 489
- .long pseudo-op 489
- .machine pseudo-op 490
- .org pseudo-op 493
- .quad pseudo-op 493
- .rename pseudo-op 495
- .set pseudo-op 496
- .short pseudo-op 497
- .source pseudo-op 498
- .space pseudo-op 499
- .stabx pseudo-op 499
- .string pseudo-op 500
- .tib pseudo-op 501
- .tc pseudo-op 503
- .toc pseudo-op 504
- .tocof pseudo-op 504
- .using pseudo-op 505
- .vbyte pseudo-op 509
- .weak pseudo-op 510
- .xline pseudo-op 511

A

- a (Add) instruction 126
- abs (Absolute) instruction 123
- accessing data through the TOC 83
- add (Add) instruction 124
- addc (Add Carrying) instruction 126
- ade (Add Extended) instruction 128
- adi (Add Immediate) instruction 130
- addic (Add Immediate Carrying) instruction 131
- addic. (Add Immediate Carrying and Record)
  instruction 132
- addid (Add Immediate Shifted) instruction 133
- addm (Add to Minus One Extended) instruction 135
- address location
  - making a translation look-aside buffer for
    using tlb (Translation Look-Aside Buffer Invalidate
    Entry) instruction 451
  - using tlb (Translation Look-Aside Buffer
    Invalidate Entry) instruction 451
- addresses
  - adding two general-purpose registers
    using add (Add) instruction 124
  - calculating from an offset less than 32KB
    using addi (Add Immediate) instruction 130
  - calculating from an offset more than 32KB
    using addis (Add Immediate Shifted)
    instruction 133
  - addressing
    - absolute 47
    - explicit-based 48
    - implicit-based 50
    - location counter 51
    - relative immediate 48
    - addze (Add to Zero Extended) instruction 137
- ae (Add Extended) instruction 128
- ai (Add Immediate) instruction 131
- ai. (Add Immediate and Record) instruction 132
- alias
  - creating for an illegal name in syntax
    using .rename pseudo-op 495
- ame (Add to Minus One Extended) instruction 135
- and (AND) instruction 138
- andc (AND with Complement) instruction 140
- andi. (AND Immediate) instruction 141
- andil. (AND Immediate Lower) instruction 141
- andis. (AND Immediate Shifted) instruction 142
- andiu. (AND Immediate Upper) instruction 142
- Appendix H: Value Definitions 595
- architecture
  - multiple hardware support 1

Numerics

32-bit applications
  - POWER family 12
  - PowerPC 12
32-bit fixed-point rotate and shift instructions
  - extended mnemonics 107
64-bit fixed-point rotate and shift instructions
  - extended mnemonics 110

© Copyright IBM Corp. 1997, 2006

607
architecture (continued)

POWER and PowerPC 11
as command 53
flags 53
assembler
features 1
interpreting a listing 59
passes 57
assembling
program 53
with the cc command 56
aze (Add to Zero Extended) instruction 137

B
b (Branch) instruction 143
base address
specifying
using .using pseudo-op 505
base register
assigning a number for
using .using pseudo-op 505
stop using specified register as
using .drop pseudo-op 476
bbf[l][a] extended mnemonic 90
bbfc[l] extended mnemonic 90
bbfr[l] extended mnemonic 90
bbt[l][a] extended mnemonic 90
bbtc[l] extended mnemonic 90
bbt[l] extended mnemonic 90
bc (Branch Conditional) instruction 144
bcc (Branch Conditional to Count Register)
instruction 147
bcctr (Branch Conditional to Count Register)
instruction 147
br[l] extended mnemonic 90
bclr (Branch Conditional Register) instruction 149
bcr (Branch Conditional Register) instruction 149
bct[l][a] extended mnemonic 90, 91
bdnl[l][a] extended mnemonic 90
bdnf[l] extended mnemonic 90
bdnz[l][a] extended mnemonic 90
bdnzf[l][a] extended mnemonic 91
bdnzfr[l] extended mnemonic 91
bdnztl[l][a] extended mnemonic 91
bdnzt[l][a] extended mnemonic 91
bdnztl[l] extended mnemonic 91
bdzl[l][a] extended mnemonic 90
bdzf[l][a] extended mnemonic 91
bdzfr[l] extended mnemonic 91
bdzfr[l] extended mnemonic 90
bdz[l][a] extended mnemonic 90
bdzflr[l] extended mnemonic 91
bf[l][a] extended mnemonic 90
bfc[l][a] extended mnemonic 91
bflr[l] extended mnemonic 91
bl (Branch and Link) instruction 143
bfr[l] extended mnemonic 91
br[l] extended mnemonic 90
branch instructions
extended mnemonics of 89
branch prediction
extended mnemonics for 93
bt(l)[a] extended mnemonic 90
bctr[l] extended mnemonic 91
bfr[l] extended mnemonic 91
C

caches
using clcs (Cache Line Compute Size)
instruction 152
using clf (Cache Line Flush) instruction 153
using cli (Cache Line Invalidate) instruction 155
using dcbf (Data Cache Block Flush)
instruction 171
using dcbi (Data Cache Block Invalidate)
instruction 172
using dcbst (Data Cache Block Store)
instruction 173
using dcbt (Data Cache Block Touch)
instruction 175
using dcbst (Data Cache Block Touch for Store)
instruction 178
using dcbz (Data Cache Block Set to Zero)
instruction 179
using dclst (Data Cache Line Store) instruction 180
using dclz (Data Cache Line Set to Zero)
instruction 179
using dcs (Data Cache Synchronize) instruction 447
using icbi (Instruction Cache Block Invalidate)
instruction 242
using ics (Instruction Cache Synchronize)
instruction 243
cal (Compute Address Lower) instruction 130
called routines 77
calling conventions
support for
pseudo-ops 465
calling routines 77
cau (Compute Address Upper) instruction 133
cax (Compute Address) instruction 124
cc command
assembling and linking with 56
caller set 27
caller values
assembling into consecutive bytes
using .string pseudo-op 500
clcs (Cache Line Compute Size) instruction 152
clf (Cache Line Flush) instruction 153
cli (Cache Line Invalidate) instruction 155
cldi extended mnemonic 112
cldsi extended mnemonic 112
cldwi extended mnemonic 109
crrdi extended mnemonic 112
crrwi extended mnemonic 109
crslwi extended mnemonic 109
cmp (Compare) instruction 156
cmpi (Compare Immediate) instruction 157
cmpl (Compare Logical) instruction 159
cmpli (Compare Logical Immediate) instruction 160
cmplw extended mnemonic 98
cmplwi extended mnemonic 98
cmpw extended mnemonic 98
cmpwi extended mnemonic 98
cntlz (Count Leading Zeros) instruction 162
cntlzd (Count Leading Zeros Double Word) Instruction 161
cntlz (Count Leading Zeros Word) instruction 162
common blocks
  defining
    using .comm pseudo-op 471
  identifying the beginning of
    using .bc pseudo-op 468
  identifying the end of
    using .ec pseudo-op 479
Condition Register 167
copying bit 3 from the Fixed-Point Exception Register into
  using mcrr (Move to Condition Register from
    XER) instruction 298
copying general-purpose register contents into
  using mtcrf (Move to Condition Register Fields)
    instruction 308
copying Summary Overflow bit from the Fixed-Point
  Exception Register into
  using mcrr (Move to Condition Register from
    XER) instruction 298
copying the Carry bit from the Fixed-Point Exception Register into
  using mcrr (Move to Condition Register from
    XER) instruction 298
copying the Overflow bit from the Fixed-Point
  Exception Register into
  using mcrr (Move to Condition Register from
    XER) instruction 298
Condition Register bit 163
placing ANDing and the complement in a Condition
  Register bit
  using crandc (Condition Register AND with Complement)
    instruction 164
placing complemented result of ANDing two
  Condition Register bits in
  using crand (Condition Register NAND)
    instruction 166
placing complemented result of XORing two
  Condition Register bits in
  using creqv (Condition Register Equivalent)
    instruction 165
placing result of ORing and complement of Condition
  Register bit in
  using crorc (Condition Register OR with Complement)
    instruction 169
placing result of ORing two Condition Register bits in
  using croc (Condition Register OR)
    instruction 168
placing result of XORing two Condition Register bits in
  using crxor (Condition Register XOR)
    instruction 170
Condition Register field
  copying the contents from one into another
    using mcrr (Move Condition Register Field)
      Instruction 296
condition register logical instructions
  extended mnemonics 96
constants 35
control sections
  giving a storage class to
    using .csect pseudo-op 473
  giving an alignment to
    using .csect pseudo-op 473
  grouping code into
    using .csect pseudo-op 473
  grouping data into
    using .csect pseudo-op 473
  naming
    using .csect pseudo-op 473
count number of one bits in doubleword 338
Count Register
  branching conditionally to address in
    using bcc (Branch Conditional to Count Register)
      instruction 147
    using bcctr (Branch Conditional to Count Register)
      instruction 147
CPU ID
  determination 4
crand (Condition Register AND) instruction 163
crandc (Condition Register AND with Complement)
  instruction 164
crclr extended mnemonic 96
creqv (Condition Register Equivalent) instruction 165
crmov extended mnemonic 96
crmnd (Condition Register NAND) instruction 166
cmoe (Condition Register) instruction 167
crnot extended mnemonic 96
cr (Condition Register OR) instruction 168
croc (Condition Register OR with Complement)
  instruction 169
cross-reference
  interpreting a symbol 63
    mnemonics 4
crset extended mnemonic 96
crxor (Condition Register XOR) instruction 170

D
data
  accessing through the TOC 83
data alignment
    pseudo-ops 464
data definition
    pseudo-ops 464
dcbf (Data Cache Block Flush) instruction 171
dcbi (Data Cache Block Invalidate) instruction 172
dcbst (Data Cache Block Store) instruction 173
dcbt (Data Cache Block Touch) instruction 175
dcbst (Data Cache Block Touch for Store)
  instruction 178
dcbz (Data Cache Block Set to Zero) instruction 179
dclst (Data Cache Line Store) instruction 180

Index 609
dclz (Data Cache Line Set to Zero) instruction 179

dcs (Data Cache Synchronize) instruction 447

debug traceback tags
  defining
    using .tbtag pseudo-op 501

debuggers
  providing information to
    using .staxb pseudo-op 499
  symbol table entries
    pseudo-ops 465
  defining
    table of contents
      using .tocof pseudo-op 504

div (Divide) instruction 182

divd (Divide Double Word) Instruction 184

divdu (Divide Double Word Unsigned) Instruction 185

divs (Divide Short) instruction 186

divw (Divide Word) instruction 188

divwu (Divide Word Unsigned) instruction 190

double floating-point constant
  storing at the next fullword location
    using .double pseudo-op 475

double-precision floating-point
  adding 64-bit operand to result of multiplying two
    operands
    using fma (Floating Multiply-Add) instruction 216, 236
    using fmadd (Floating Multiply-Add)
      instruction 216, 236
  adding two 64-bit operands
    using fa (Floating Add) instruction 203
    using fadd (Floating Add Double) instruction 203
  dividing 64-bit operands
    using fd (Floating Divide) instruction 213
    using fdv (Floating Divide Double)
      instruction 213
  multiplying two 64-bit operands
    using fm (Floating Multiply) instruction 221
    using fmul (Floating Multiply Double)
      instruction 221
  multiplying two 64-bit operands and adding to 64-bit
    operand
    using fnma (Floating Negative Multiply-Add)
      instruction 226
    using fnmadd (Floating Negative Multiply-Add Double)
      instruction 226
  multiplying two 64-bit operands and subtracting 64-bit
    operand
    using fnms (Floating Negative Multiply-Subtract)
      instruction 228
    using fnmsub (Floating Negative Multiply-Subtract Double)
      instruction 228
  rounding 64-bit operand to single precision
    using frsp (Floating Round to Single Precision)
      instruction 233
  subtracting 64-bit operand from result of multiplying
    two 64-bit operands
    using fs (Floating Subtract) instruction 240
    using fsm (Floating Subtract)
      instruction 219
    using fsub (Floating Subtract Double)
      instruction 240
  subtracting 64-bit operands
    using fs (Floating Subtract) instruction 240
    using fsub (Floating Subtract Double)
      instruction 240

doz (Difference or Zero) instruction 191

dodzi (Difference or Zero Immediate) instruction 193

dummy control sections
  identifying the beginning of
    using .dsect pseudo-op 477
  identifying the continuation of
    using .dsect pseudo-op 477

e
eciwx (External Control In Word Indexed) instruction 194
ecowx (External Control Out Word Indexed) instruction 195
eieio (Enforce In-Order Execution of I/O) instruction 196
epilogs 70
  actions 71
eqv (Equivalent) instruction 198
error
  messages 513
error conditions
  detection 6
expressions 39
assembling into a TOC entry
  using .tc pseudo-op 503
assembling into consecutive bytes 470
assembling into consecutive double-words
  using .long pseudo-op 489
assembling into consecutive fullwords
  using .long pseudo-op 489
assembling into consecutive halfwords
  using .short pseudo-op 497
assembling the value into consecutive bytes
  using .vbyte pseudo-op 509
facilitating the use of local symbols in
  using .tocof pseudo-op 504
setting a symbol equal in type and value to
  using .set pseudo-op 496
extended mnemonics
  for branch prediction 93
    of 32-bit fixed-point rotate and shift instructions 107
    of 64-bit fixed-point rotate and shift instructions 110
    of branch instructions 89
    of condition register logical instructions 96
    of fixed-point arithmetic instructions 97
    of fixed-point compare instructions 98
    of fixed-point load instructions 99
    of fixed-point logical instructions 100
    of fixed-point trap instructions 100
    of moving from or to special-purpose registers 102

external symbol definitions
  pseudo-ops 464
extldi extended mnemonic 112
extlwi extended mnemonic 109
extrdi extended mnemonic 112

610  Assembler Language Reference
extrwi extended mnemonic 109
exts (Extend Sign) instruction 200
extsb (Extend Sign Byte) instruction 199
extsh (Extend Sign Halfword) instruction 200
extsw (Extend Sign Word) Instruction 197

fa (Floating Add) instruction 203
fabs (Floating Absolute Value) instruction 202
fadd (Floating Add Double) instruction 203
fadds (Floating Add Single) instruction 203
fcfid (Floating Convert from Integer Double Word) Instruction 205
fcir (Floating Convert Double to Integer with Round) instruction 210
fcirz (Floating Convert Double to Integer with Round to Zero) instruction 212
fcmpo (Floating Compare Ordered) instruction 206
fcmpu (Floating Compare Unordered) instruction 207
fctid (Floating Convert to Integer Double Word) Instruction 208
fctidz (Floating Convert to Integer Double Word with Round toward Zero) Instruction 209
fctiw (Floating Convert to Integer Word) instruction 210
fctiwz (Floating Convert to Integer Word with Round to Zero) instruction 212
fd (Floating Divide) instruction 213
fdiv (Floating Divide Double) instruction 213
fdivs (Floating Divide Single) instruction 213
fixed-point arithmetic instructions
extended mnemonics 97
fixed-point compare instructions
extended mnemonics 98
fixed-point load instructions 99
fixed-point logical instructions
extended mnemonics 100
fixed-point trap instructions 100
floating-point constants
storing at the next fullword location
using .float pseudo-op 483
floating-point numbers 24
floating-point registers
calculating a square root
using fsqrt (Floating Square Root) instruction 231, 235
comparing contents of two
using fcmpeq (Floating Compare Ordered) instruction 206
using fcmpeq (Floating Compare Unordered) instruction 207
converting 64-bit double-precision floating-point operand
using fcir (Floating Convert to Integer with Round) instruction 210
using fcirz (Floating Convert Double to Integer with Round to Zero) instruction 212
using fctiw (Floating Convert to Integer Word) instruction 210
using fctiwz (Floating Convert to Integer Word with Round to Zero) instruction 212
floating-point registers (continued)
converting contents to single precision
stfsx (Store Floating-Point Single Indexed) instruction 417
using stfs (Store Floating-Point Single) instruction 414
using stfsu (Store Floating-Point Single with Update) instruction 415
using stfsux (Store Floating-Point Single with Update Indexed) instruction 416
copying contents into Floating-Point Status and Control Register
using mtsfs (Move to FPSCR Fields) instruction 311
interpreting the contents of 25
loading converted double-precision floating-point number into
using lfs (Load Floating-Point Single) instruction 262
using lfsu (Load Floating-Point Single with Update) instruction 263
using lfsux (Load Floating-Point Single with Update Indexed) instruction 264
loading doubleword of data from memory into
using lfd (Load Floating-Point Double) instruction 253
using lfdw (Load Floating-Point Double with Update) instruction 254
using lfdwx (Load Floating-Point Double with Update Indexed) instruction 255
using lfdx (Load Floating-Point Double Indexed) instruction 256
loading quadword of data from memory into
using lfq (Load Floating-Point Quad) instruction 257
using lfqu (Load Floating-Point Quad with Update) instruction 258
using lfqux (Load Floating-Point Quad with Update Indexed) instruction 260
using lfx (Load Floating-Point Quad Indexed) instruction 261
moving contents of to another
using fmr (Floating Move Register) instruction 218
negating absolute contents of
using fnabs (Floating Negative Absolute Value) instruction 224
negating contents of
using fneg (Floating Negative) instruction 225
storing absolute value of contents into another
using fabs (Floating Absolute Value) instruction 202
storing contents into doubleword storage
using stfd (Store Floating-Point Double) instruction 405
using stfdw (Store Floating-Point Double with Update) instruction 406
using stfdwx (Store Floating-Point Double with Update Indexed) instruction 407
using stfdx (Store Floating-Point Double Indexed) instruction 408
floating-point registers (continued)
storing contents into quadword storage
  using stfq (Store Floating-Point Quad) instruction 410
  using stfq (Store Floating-Point Quad with Update) instruction 411
  using stfqux (Store Floating-Point Quad with Update Indexed) instruction 412
  using stfqx (Store Floating-Point Quad Indexed) instruction 413
storing contents into word storage
  using stfiwx (Store Floating-Point as Integer word Indexed) instruction 409
Floating-Point Status and Control Register
  copying an immediate value into a field of
    using mtfsfi (Move to FPSCR Field Immediate) instruction 313
  copying the floating-point register contents into
    using mtfsf (Move to FPSCR Fields) instruction 311
  filling the upper 32 bits after loading
    using mtfsf (Move from FPSCR) instruction 299
  loading contents into a floating-point register
    using mtfsf (Move from FPSCR) instruction 299
  setting a specified bit to 1
    using mtfsb1 (Move to FPSCR Bit 1) instruction 310
  setting a specified bit to zero
    using mtfsb0 (Move to FPSCR Bit 0) instruction 309
Floating-Point Status and Control Register field
  copying the bits into the Condition Register
    using mcrfs (Move to Condition Register from FPSCR) instruction 296
func (Floating Multiply) instruction 221
fma (Floating Multiply-Add) instruction 216
fmadd (Floating Multiply-Add Double) instruction 216
fmadds (Floating Multiply-Add Single) instruction 216
fmr (Floating Move Register) instruction 218
fms (Floating Multiply-Subtract) instruction 219
fmsub (Floating Multiply-Subtract Double) instruction 219
fmsubs (Floating Multiply-Subtract Single) instruction 219
fmul (Floating Multiply-Subtract Single) instruction 219
fnabs (Floating Negative Absolute Value) instruction 221
fneg (Floating Negative) instruction 225
fnma (Floating Negative Multiply-Add) instruction 226
fmadd (Floating Negative Multiply-Add Double) instruction 226
fmadds (Floating Negative Multiply-Add Single) instruction 226
fnms (Floating Negative Multiply-Subtract Double) instruction 226
fnmsub (Floating Negative Multiply-Subtract Single) instruction 226
fnmsubs (Floating Negative Multiply-Subtract Double) instruction 226
fres (Floating Reciprocal Estimate) instruction 231
frsp (Floating Round to Single Precision) instruction 233
fresqte (Floating Reciprocal Square Root Estimate) instruction 235
fs (Floating Subtract) instruction 240
fser (Floating-Point Select) instruction 236
fsqrt (Floating Square Root, Double-Precision) Instruction 237
fsqrt (Floating Square Root Single) Instruction 239
fsub (Floating Subtract Double) instruction 240
fsub (Floating Subtract Single) instruction 240
functions
  identifying
    using .function pseudo-op 483
  identifying the beginning of
    using .bf pseudo-op 468
  identifying the end of
    using .ef pseudo-op 480
G
general-purpose registers
  adding complement from -1 with carry
    using sfme (Subtract from Minus One Extended) instruction 442
    using subfme (Subtract from Minus One Extended) instruction 442
  adding contents to the value of the Carry bit
    using adde (Add Extended) instruction 128
    using ae (Add Extended) instruction 128
  adding contents with 16-bit signed integer
    using addic (Add Immediate Carrying) instruction 131
    using ai (Add Immediate) instruction 131
  adding contents with Carry bit and -1
    using addme (Add to Minus One Extended) instruction 135
    using ame (Add to Minus One Extended) instruction 135
  adding immediate value to contents of
    using addic (Add Immediate Carrying and Record) instruction 132
    using ai (Add Immediate and Record) instruction 132
  adding the complement of the contents with the Carry bit
    using sfze (Subtract from Zero Extended) instruction 444
    using subfze (Subtract from Zero Extended) instruction 444
  adding the contents of
    using adde (Add Carrying) instruction 126
  adding zero and the value of the Carry bit to the contents of
    using addze (Add to Zero Extended) instruction 137
    using aze (Add to Zero Extended) instruction 137
general-purpose registers (continued)

ANDing a generated mask with the rotated contents of

using rlrm (Rotated Left Immediate Then AND with Mask) instruction 354
using rlrm (Rotate Left Then AND with Mask) instruction 356
using rlwinn (Rotated Left Word Immediate Then AND with Mask) instruction 354
using rlwinn (Rotate Left Word Then AND with Mask) instruction 356

ANDing an immediate value with

using andi. (AND Immediate) instruction 141
using andil. (AND Immediate Lower) instruction 141

ANDing contents with the complement of another

using andc (AND with Complement) instruction 140

ANDing logically the contents of

using and (AND) instruction 138

ANDing most significant 16 bits with a 16-bit unsigned integer

using andis. (AND Immediate Shifted) instruction 142
using andiu. (AND Immediate Upper) instruction 142

changing the arithmetic sign of the contents of

using neg (Negate) instruction 331

comparing contents algebraically

using cmp (Compare) instruction 156

comparing contents logically

using cmpl (Compare Logical) instruction 159

comparing contents with unsigned integer logically

using cmpli (Compare Logical Immediate) instruction 160

comparing contents with value algebraically

using cmpi (Compare Immediate) instruction 157

computing difference between contents and signed 16-bit integer

using dozi (Difference or Zero Immediate) instruction 193

computing difference between contents of two

using doz (Difference or Zero) instruction 191

copying bit 0 of halfword into remaining 16 bits

using lha (Load Half Algebraic) instruction 266

copying bit 0 of halfword into remaining 16 bits of

using lhau (Load Half Algebraic with Update) instruction 267

using lhaux (Load Half Algebraic with Update Indexed) instruction 268

using lhax (Load Half Algebraic Indexed) instruction 269

copying Condition Register contents into

using mfcr (Move from Condition Register) instruction 298
using mfocrf (Move from One Condition Register Field) instruction 301
using mtocrf (Move to One Condition Register Field) instruction 314

copying contents into a special-purpose register

using mtspr (Move to Special-Purpose Register) instruction 315

copying contents into the Condition Register

using mtcrf (Move to Condition Register Fields) instruction 308

copying special-purpose register contents into

using mfspr (Move from Special-Purpose Register) instruction 303

copying the Machine State Register contents into

using mfmsr (Move from Machine State Register) instruction 300

dividing by contents of

using div (Divide) instruction 182
using divs (Divide Short) instruction 186

generating mask of ones and zeros for loading into

using maskg (Mask Generate) instruction 293

inserting contents of one into another under bit-mask control

using maskir (Mask Insert from Register) instruction 294

loading consecutive bytes from memory into consecutive

using isi (Load String Immediate) instruction 280
using iswi (Load String Word Immediate) instruction 280
using lswwx (Load String Word Indexed) instruction 282

using lsx (Load String Indexed) instruction 282

loading consecutive bytes into

using lscbx (Load String and Compare Byte Indexed) instruction 278

loading consecutive words into several

using im (Load Multiple) instruction 276
using imw (Load Multiple Word) instruction 276

loading word of data from memory into

using lu (Load with Update) instruction 289
using lwzu (Load Word with Zero Update) instruction 289

loading word of data into

using lux (Load with Update Indexed) instruction 291

using lwzux (Load Word and Zero with Update Indexed) instruction 291

using lwzx (Load Word and Zero Indexed) instruction 292

using lx (Load Indexed) instruction 292

logically complementing the result of ANDing the contents of two

using nand (NAND) instruction 330

logically complementing the result of ORing the contents of two

using nor (NOR) instruction 333

logically ORing the content of two

using or (OR) instruction 334

logically ORing the contents with the complement of the contents of

using orc (OR with Complement) instruction 335
general-purpose registers  *(continued)*
merging a word of zeros with the MQ Register contents
  using srlq (Shift Right Long with MQ) instruction 392
merging rotated contents with a word of 32 sign bits
  using sra (Shift Right Algebraic) instruction 380
  using srai (Shift Right Algebraic Immediate) instruction 382
  using sraiq (Shift Right Algebraic Immediate with MQ) instruction 377
  using sraq (Shift Right Algebraic with MQ) instruction 378
  using sraw (Shift Right Algebraic Word) instruction 380
  using srawi (Shift Right Algebraic Word Immediate) instruction 382
merging rotated contents with the MQ Register contents
  using sreq (Shift Right Extended with MQ) instruction 387
  using srlq (Shift Right Long Immediate with MQ) instruction 390
  using srlq (Shift Right Long with MQ) instruction 392
merging the rotated contents results with the MQ Register contents
  using slliq (Shift Left Long Immediate with MQ) instruction 369
merging with masked MQ Register contents
  using sleq (Shift Left Extended with MQ) instruction 366
multiplying a word
  using mulhw (Multiply High Word) instruction 321
  using mulhwu (Multiply High Word Unsigned) instruction 322
multiplying the contents by a 16-bit signed integer
  using muli (Multiply Immediate) instruction 325
  using muli (Multiply Low Immediate) instruction 325
multiplying the contents of two
  using mul (Multiply) instruction 317
multiplying the contents of two general-purpose registers into
  using mullw (Multiply Low Word) instruction 326
  using muls (Multiply Short) instruction 326
negating the absolute value of
  using nabs (Negative Absolute) instruction 328
ORing the lower 16 bits of the contents with a 16-bit unsigned integer
  using ori (OR Immediate) instruction 336
  using ori (OR Immediate Lower) instruction 336
ORing the upper 16 bits of the contents with a 16-bit unsigned integer
  using oris (OR Immediate Shifted) instruction 337
  using oriu (OR Immediate Upper) instruction 337
placing a copy of rotated contents in the MQ Register
  using srea (Shift Right Extended Algebraic) instruction 386
placing a copy of rotated data in the MQ register
  using sile (Shift Left Extended) instruction 364
placing number of leading zeros in
  using cntlz (Count Leading Zeros) instruction 162
  using cntlzw (Count Leading Zeros Word) instruction 162
placing rotated contents in the MQ Register
  using sliq (Shift Left Immediate with MQ) instruction 367
  using sraq (Shift Right Algebraic with MQ) instruction 378
  using sraw (Shift Right Algebraic Word) instruction 380
  using srawi (Shift Right Algebraic Word Immediate) instruction 382
placing the absolute value of the contents in
  using abs (Absolute) instruction 123
placing the logical AND of the rotated contents in
  using srq (Shift Right with MQ) instruction 393
placing the rotated contents in the MQ register
  using sreq (Shift Right with MQ) instruction 393
rotating contents left
  using rlmi (Rotate Left Then Mask Insert) instruction 350
  using sl (Shift Left) instruction 373
  using sliq (Shift Left Immediate with MQ) instruction 367
  using slliq (Shift Left Long Immediate with MQ) instruction 369
  using sr (Shift Right) instruction 395
  using sra (Shift Right Algebraic) instruction 380
  using sraq (Shift Right Algebraic with MQ) instruction 378
  using srea (Shift Right Extended Algebraic) instruction 386
  using sreq (Shift Right Extended with MQ) instruction 387
  using srq (Shift Right Algebraic with MQ) instruction 378
setting remaining 16 bits to 0 after loading
  using lhz (Load Half and Zero) instruction 272
setting remaining 16 bits to zero after loading
  using lhzu (Load Half and Zero with Update) instruction 273
  using lhbrx (Load Half Byte-Reverse Indexed) instruction 270
setting remaining 16 bits to zero in
  using lhbx (Load Half Byte-Reverse Indexed) instruction 274
storing a byte into memory with the address in
  using sbu (Store Byte with Update) instruction 397
storing a byte of data into memory
  using stb (Store Byte) instruction 396
  using stbux (Store Byte with Update Indexed) instruction 398
  using stbx (Store Byte Indexed) instruction 399
general-purpose registers

storing a byte-reversed word of data into memory
using stbrx (Store Byte Reverse Indexed) instruction 429
using stwbrx (Store Word Byte Reverse Indexed) instruction 429

storing a halfword of data into memory
using sth (Store Half) instruction 418
using sthu (Store Half with Update) instruction 420
using sthux (Store Half with Update Indexed) instruction 421
using sthx (Store Half Indexed) instruction 422

storing a word of data into memory
using st (Store) instruction 428
using stu (Store with Update) instruction 431
using stux (Store with Update Indexed) instruction 433
using stw (Store Word) instruction 428
using stwcx (Store Word Conditional Indexed) instruction 430
using stwu (Store Word with Update) instruction 431
using stwx (Store Word with Update Indexed) instruction 433
using stwx (Store Word Indexed) instruction 434
using stx (Store Indexed) instruction 434

storing consecutive bytes from consecutive registers into memory
using stsl (Store String Immediate) instruction 425
using stswi (Store String Word Immediate) instruction 425
using stswx (Store String Word Indexed) instruction 430
using stswx (Store String Indexed) instruction 434

storing contents of consecutive registers into memory
using stm (Store Multiple) instruction 423
using stmw (Store Multiple Word) instruction 423

storing halfword of data with 2 bytes reversed into memory
using sthbrx (Store Half Byte-Reverse Indexed) instruction 419

subtracting contents of one from another
using sf (Subtract From) instruction 437
using subfc (Subtract From Carrying) instruction 437

subtracting from
using subf (Subtract From) instruction 435
subtracting the contents from a 16-bit signed integer
using sfi (Subtract from Immediate) instruction 441
using subfc (Subtract from Immediate Carrying) instruction 441

subtracting the contents from the sum of
using sfe (Subtract from Extended) 439
using subfe (Subtract from Extended) 439
subtracting the value of a signed integer from the contents of
using si (Subtract Immediate) instruction 362
subtracting the value of a signed integer from the contents of
using si. (Subtract Immediate and Record) instruction 363

translate effective address into real address and store in
using rac (Real Address Compute) instruction 339
using a (Add) instruction 126
using divw (Divide Word) instruction 188
using divwu (Divide Word Unsigned) instruction 190
using extsb (Extend Sign Byte) instruction 199
using l (Load Floating-Point Quad) instruction 257
using lfuq (Load Floating-Point Quad with Update) instruction 258
using lfuqx (Load Floating-Point Quad with Update Indexed) instruction 260
using lfu (Load Floating-Point Quad Indexed) instruction 261
using lwarx (Load Word and Reserve Indexed) instruction 284
using r (Rotate Left Immediate Then Mask Insert) instruction 352
using rlnm (Rotate Left Then AND with Mask) instruction 356
using rlnm (Rotate Left Then AND with Mask) instruction 356
using rlla (Rotate Left and Insert Bit) instruction 358
using rll (Shift Left Long with MQ) instruction 370
using rll (Shift Left with MQ) instruction 372
using rll (Shift Left Word) instruction 373
using rai (Shift Right Algebraic Immediate) instruction 382
using raiq (Shift Right Algebraic Immediate with MQ) instruction 377
using rraw (Shift Right Algebraic Word) instruction 380
using rraw (Shift Right Algebraic Word) instruction 380
using rrawi (Shift Right Algebraic Word Immediate) instruction 382
using rrawi (Shift Right Algebraic Word Immediate) instruction 382
using rrawi (Shift Right Extended) instruction 384
using rrawi (Shift Right Long Immediate with MQ) instruction 390
using rrawi (Shift Right Long with MQ) instruction 392
using rrawi (Shift Right with MQ) instruction 393
using rrawi (Shift Right Word) instruction 395
using rrawi (Shift Right Word) instruction 395
using rraw (Store Floating-Point Quad) instruction 410
using rraw (Store Floating-Point Quad with Update) instruction 411
using rraw (Store Floating-Point Quad with Update Indexed) instruction 412
using rraw (Store Floating-Point Quad Indexed) instruction 413
XORing contents of
using eqv (Equivalent) instruction 198
general-purpose registers (continued)
XORing the contents and 16-bit unsigned integer
using xori (XOR Immediate) instruction 459
using xoril (XOR Immediate Lower) instruction 459
XORing the contents of two
using xor (XOR) instruction 458
XORing the upper 16 bits with a 16-bit unsigned integer
using xoris (XOR Immediate Shift) instruction 460
using xoriu (XOR Immediate Upper) instruction 460

H
hash values
associating with external symbol
using .hash pseudo-op 485
host machine independence 4

I
icbi (Instruction Cache Block Invalidate) instruction 242
ics (Instruction Cache Synchronize) instruction 243
implementation
multiple platform support 1
included files
identifying the beginning of
using .bi pseudo-op 469
identifying the end of
using .ei pseudo-op 480
inner blocks
identifying the beginning of
using .bb pseudo-op 467
identifying the end of
using .eb pseudo-op 479
inslwi extended mnemonic 109
insrdi extended mnemonic 112
insrwi extended mnemonic 109
installing the assembler 9
instruction fields 16
instruction forms 13
instructions
branch 19
common to POWER and PowerPC 561
condition register 20
fixed-point
address computation 22
arithmetic 22
compare 22
load and store 21
load and store with update 21
logical 23
move to or from special-purpose registers 23
rotate and shift 23
string 22
trap 22
floating-point
arithmetic 25
compare 26
instructions (continued)
floating-point (continued)
conversion 26
load and store 25
move 25
multiply-add 25
status and control register 26
PowerPC 575
PowerPC 601 RISC Microprocessor 585
sorted by mnemonic 533
sorted by primary and extended op code 547
system call 20
intermodule calls using the TOC 85
interrupts
generating when a condition is true
using t (Trap) instruction 456
using ti (Trap Immediate) instruction 457
using tw (Trap Word) instruction 456
using twi (Trap Word Immediate) instruction 457
supervisor call
generating an interrupt 446
system call
generating an interrupt 360
system call vectored
generating an interrupt 361
isync (Instruction Synchronize) instruction 243

L
l (Load) instruction 288
la extended mnemonic 99
lbx (Load Byte-Reverse Indexed) instruction 287
lbz (Load Byte and Zero) instruction 245
lbzux (Load Byte and Zero with Update Indexed) instruction 247
lbzx (Load Byte and Zero Indexed) instruction 248
ld (Load Double Word) instruction 249
ldarx (Store Double Word Reserve Indexed) Instruction 250
lddu (Load Double Word with Update) instruction 251
ldq (Load Floating-Point Quad) instruction 252
ldqdu (Load Floating-Point Quad with Update) instruction 253
ldqx (Load Floating-Point Quad Indexed) instruction 254
leading zeros
placing in a general-purpose register
using cntlz (Count Leading Zeros) instruction 162
using cntlzw (Count Leading Zeros Word) instruction 162
lf (Load Floating-Point) instruction 255
lfdu (Load Floating-Point Double with Update) instruction 256
lfdux (Load Floating-Point Double with Update Indexed) instruction 257
lfq (Load Floating-Point Quad) instruction 258
lfqu (Load Floating-Point Quad with Update) instruction 260
lfqux (Load Floating-Point Quad with Update Indexed) instruction 261
lfqx (Load Floating-Point Quad Indexed) instruction 261
lfs (Loading Floating-Point Single) instruction 262
lfsu (Load Floating-Point Single with Update) instruction 263
lfsux (Load Floating-Point Single with Update Indexed) instruction 264
lfsx (Load Floating-Point Single Indexed) instruction 265
lha (Load Half Algebraic) instruction 266
lhau (Load Half Algebraic with Update) instruction 267
lhaux (Load Half Algebraic with Update Indexed) instruction 268
lhax (Load Half Algebraic Indexed) instruction 269
lhbrx (Load Half Byte-Reverse Indexed) instruction 270
lhz (Load Half and Zero) instruction 272
lhzu (Load Half and Zero with Update) instruction 273
lhzux (Load Half and Zero with Update Indexed) instruction 274
lhzx (Load Half and Zero Indexed) instruction 275
li extended mnemonic 99
lil extended mnemonic 99
line format 28
line numbers identifying
using .line pseudo-op 488
lines representing the number of
using .xline pseudo-op 511
Link Register
branching conditionally to address in
using bcr (Branch Conditional Register) instruction 149
using bcr (Branch Conditional Register) instruction 149
linkage subroutine linkage convention 65
linker making a symbol globally visible to the using .globl pseudo-op 484
linking 53
with the cc command 56
lis extended mnemonic 99
listing interpreting an assembler 59
liu extended mnemonic 99
lm (Load Multiple) instruction 276
lmw (Load Multiple Word) instruction 276
local common section defining a
using .comm pseudo-op 486
local symbol facilitating use in expressions
using .tocof pseudo-op 504
location counter 51
advancing until a specified boundary is reached
using .align pseudo-op 466
setting the value of the current
using .org pseudo-op 493
logical processing model 11
lq (Load Quad Word) instruction 277
lsbcb (Load String and Compare Byte Indexed) instruction 278
lsi (Load String Immediate) instruction 280
lswi (Load String Word Immediate) instruction 280
lswx (Load String Word Indexed) instruction 282
lsx (Load String Indexed) instruction 282
lu (Load with Update) instruction 289
lux (Load with Update Indexed) instruction 291
lwa (Load Word Algebraic) instruction 284
lwax (Load Word Algebraic Indexed) instruction 286
lwz (Load Word and Zero) instruction 288
lwzu (Load Word with Zero Update) instruction 289
lwzux (Load Word and Zero with Update Indexed) instruction 291
lwzx (Load Word and Zero Indexed) instruction 292
lx (Load Indexed) instruction 292
M Machine State Register
after a supervisor call and reinitialize
using rfsvc (Return from SVC) instruction 342
continue processing and reinitialize
using rfi (Return from Interrupt) instruction 341
copying the contents into a general-purpose register
using mfmsr (Move from Machine State Register) instruction 300
main memory ensuring storage access in
using eieio (Enforce In-Order Execution of I/O) instruction 196
maskg (Mask Generate) instruction 293
maskir (Mask Insert from Register) instruction 294
masks generating instance of ones and zeros
using maskg (Mask Generate) instruction 293
mcrf (Move Condition Register Field) instruction 296
mcrfs (Move to Condition Register from FPSCR) instruction 296
mcrxr (Move to Condition Register from XER) instruction 298
memory loading a byte of data from
using lbzu (Load Byte and Zero with Update) instruction 246
loading byte of data from
using lbz (Load Byte and Zero) instruction 245
using lbzux (Load Byte and Zero with Update Indexed) instruction 247
loading byte of data into
using lbzx (Load Byte and Zero Indexed) instruction 248
Index 617
memory (continued)

loading byte-reversed halfword of data from
using lhbrx (Load Half Byte-Reverse Indexed)
  instruction 270
loading byte-reversed word of data from
using lbx (Load Byte-Reverse Indexed)
  instruction 287
using lwbrx (Load Word Byte-Reverse Indexed)
  instruction 287
loading consecutive bytes from
using lsi (Load String Immediate) instruction 280
using lswi (Load String Word Immediate)
  instruction 280
using lsxw (Load String Indexed)
  instruction 282
using lswx (Load String Indexed) instruction 282
loading doubleword of data from
using lfd (Load Floating-Point Double)
  instruction 253
using lfdx (Load Floating-Point Double Indexed)
  instruction 256
loading halfword of data from
using lha (Load Half Algebraic) instruction 266
using lhau (Load Half Algebraic with Update)
  instruction 267
using lhaux (Load Half Algebraic with Update
Indexed) instruction 268
using lhax (Load Half Algebraic Indexed)
  instruction 269
using lhz (Load Half and Zero) instruction 272
using lhzu (Load Half and Zero with Update)
  instruction 273
using lhzux (Load Half and Zero with Update
Indexed) instruction 274
using lhzx (Load Half and Zero Indexed)
  instruction 275
loading quadword of data from
using lfq (Load Floating-Point Quad)
  instruction 257
using lfx (Load Floating-Point Quad Indexed)
  instruction 261
loading single-precision floating-point number from
using lfs (Load Floating-Point Single with
Update) instruction 263
using lfsx (Load Floating-Point Single Indexed)
  instruction 265
loading single-precision floating-point number into
using lfs (Load Floating-Point Single)
  instruction 262
using lfsux (Load Floating-Point Single with
Update Indexed) instruction 264
loading word of data from
using lu (Load with Update) instruction 289
loading word of data from (continued)
using lux (Load with Update Indexed)
  instruction 291
using lvzu (Load Word with Zero Update)
  instruction 289
using lvzux (Load Word and Zero with Update
Indexed) instruction 291
using lvzx (Load Word and Zero Indexed)
  instruction 292
setting remaining 24 bits after loading into
using lbzx (Load Byte and Zero Indexed)
  instruction 248
setting remaining 24 bits to 0 after loading from
using lbz (Load Byte and Zero) instruction 245
using lbzux (Load Byte and Zero with Update
Indexed) instruction 247
setting remaining 24 bits to 0 after loading into
using lbzu (Load Byte and Zero with Update)
  instruction 246
storing a quadword of data into
using stfq (Store Floating-Point Quad)
  instruction 410
using stfqx (Store Floating-Point Quad Indexed)
  instruction 413
using dcbf (Data Cache Block Flush)
  instruction 171
messages
error 513
warning 513
mfcr (Move from Condition Register) instruction 298
mfctre extended mnemonic 106
mfdr extended mnemonic 106
mfx extended mnemonic 106
mfdsr extended mnemonic 106
mfem extended mnemonic 106
mftsr extended mnemonic 106
mfmsr (Move from Machine State Register)
  instruction 300
mfoe (Move from One Condition Register Field)
  instruction 301
mfptr extended mnemonic 106
mfptrc extended mnemonic 106
mfptrcu extended mnemonic 106
mfsdrl extended mnemonic 106
mfspr (Move from Special-Purpose Register)
  instruction 303
mfsprg extended mnemonic 106
mfsr (Move from Segment Register) instruction 305
mfsri (Move from Segment Register Indirect)
  instruction 306
mfsrin (Move from Segment Register Indirect)
  instruction 307
mfsrir extended mnemonic 106
mfsrr1 extended mnemonic 106
mfxcnr extended mnemonic 106
milicode routines 80
mnemonic
  instructions sorted by 533
mnemonics cross-reference 4
moving from or to special-purpose registers
  extended mnemonics 102
mr (Move Register) instruction 334
mr[.] extended mnemonic 100
mtcrf (Move to Condition Register Fields) instruction 308
mtctcr extended mnemonic 107
mtctdr extended mnemonic 107
mtcitsr extended mnemonic 107
mtctear extended mnemonic 107
mtfsb0 (Move to FPSCR Bit 0) instruction 309
mtfsb1 (Move to FPSCR Bit 1) instruction 310
mtfsf (Move to FPSCR Fields) instruction 311
mtfsfi (Move to FPSCR Field Immediate) instruction 313
mtlt extended mnemonic 107
mtmtr extended mnemonic 107
mtocrf (Move to One Condition Register Field) instruction 314
mtrtc1 extended mnemonic 107
mtrtcu extended mnemonic 107
mtspr0 extended mnemonic 107
mtspr extended mnemonic 107
mul (Multiply) instruction 317
mulhd (Multiply High Double Word) Instruction 319
mulhdu (Multiply High Double Word Unsigned) Instruction 320
mulhw (Multiply High Word) instruction 321
mulhwu (Multiply High Word Unsigned) instruction 322
mul (Multiply Immediate) instruction 325
mulld (Multiply Low Double Word) Instruction 324
mulldo (Multiply Low Double Word) Instruction 325
mullli (Multiply Low Immediate) instruction 326
mulldi (Multiply Low Immediate) instruction 327
mulw (Multiply Low Word) instruction 326
mulsw (Multiply Short) instruction 326

N

nabs (Negative Absolute) instruction 328
name
  creating a synonym or alias for an illegal name using :rename pseudo-op 495
nand (NAND) instruction 330
neg (Negate) instruction 331
nop extended mnemonic 100
nor (NOR) instruction 333
not[.] extended mnemonic 100
notational conventions
  pseudo-ops 466

O

op code
  instructions sorted by primary and extended 547
operators 38
or (OR) instruction 334
orc (OR with Complement) instruction 335
ori (OR Immediate) instruction 336
oril (OR Immediate Lower) instruction 336
oris (OR Immediate Shifted) instruction 337
oriu (OR Immediate Upper) instruction 337
output file
  skipping a specified number of bytes in using .space pseudo-op 499

P

passes
  assembler 57
popcntbd (Population Count Byte Doubleword) 338
POWER and POWER2
  instructions 565
POWER and PowerPC
    architecture 11
    common instructions 561
    functional differences for 114
    PowerPC instructions 120
    with same op code 115
PowerPC
  instructions 575
PowerPC 601 RISC Microprocessor
  instructions 585
PowerPC instructions
  added 120
process
  runtime process stack 67
program
  running a 87
programs
  generating interrupt
    using t (Trap) instruction 456
    using ti (Trap Immediate) instruction 457
    using tw (Trap Word) instruction 456
    using twi (Trap Word Immediate) instruction 457
prologs 70
  actions 71
  addressing 464
  calling conventions
    support for 465
data alignment 464
    functional groups 463
miscellaneous 465
  support for calling conventions 465
symbol table entries for debuggers 465
Q
quad floating-point constant
  storing at the next fullword location
  using .quad pseudo-op 493

R
rac (Real Address Compute) instruction 339
real address
  translating effective address to
  using ecixw (External Control In Word Indexed)
    instruction 194
  using ecowx (External Control Out Word Indexed)
    instruction 195
reciprocal, floating single estimate 231
reciprocal, floating square root estimate 235
ref pseudo-op 494
registers
  special-purpose
    changes and field handling 8
    extended mnemonics 102
  usage and conventions 65
reserved words 28
rfi (Return from Interrupt) instruction 341
rfid (Return from Interrupt Double Word)
  Instruction 341
rsvc (Return from SVC) instruction 361
rldcl (Rotate Left Double Word then Clear Left)
  Instruction 343
rldcr (Rotate Left Double Word then Clear Right)
  Instruction 345
rldic (Rotate Left Double Word Immediate then Clear)
  Instruction 346
rldicl (Rotate Left Double Word Immediate then Clear Left)
  Instruction 344, 347
rldicr (Rotate Left Double Word Immediate then Clear Right)
  Instruction 348
rldimi (Rotate Left Double Word Immediate then Mask Insert)
  Instruction 349
rlimi (Rotate Left Immediate Then Mask Insert)
  Instruction 352
rlimn (Rotate Left Immediate Then AND with Mask)
  instruction 354
rlmi (Rotate Left Then Mask Insert) instruction 350
rlmn (Rotate Left Then AND with Mask) instruction 356
rlwimi (Rotate Left Word Immediate Then Mask Insert)
  Instruction 352
rlwinm (Rotate Left Word Immediate Then AND with Mask)
  instruction 354
rlwm (Rotate Left Word Then AND with Mask)
  instruction 356
rotld extended mnemonic 112
rotldi extended mnemonic 112
rotlw extended mnemonic 109
rotlwi extended mnemonic 109
rotrdi extended mnemonic 112
rotrwi extended mnemonic 109
rrb (Rotate Right and Insert Bit) instruction 358
running a program 87

S
sc (System Call) instruction 360
scv (System Call Vectored) instruction 361
section definition
  pseudo-ops 464
Segment Register
  copying to general-purpose registers
    using mfsr (Move from Segment Register)
      instruction 305
    using mfsri (Move from Segment Register Indirect)
      instruction 306
    using mfsrin (Move from Segment Register Indirect) instruction 307
selecting operand with fsel instruction 236
sf (Subtract from) instruction 437
sfe (Subtract from Extended) instruction 439
sfi (Subtract from Immediate) instruction 441
sfme (Subtract from Minus One Extended)
  instruction 442
sfze (Subtract from Zero Extended) instruction 444
si (Subtract Immediate) instruction 362
si. (Subtract Immediate and Record) instruction 363
si[,] extended mnemonic 97
signed integers
  extending 16-bit to 32 bits
    using exts (Extend Sign) instruction 200
    using extsh (Extend Sign Halfword)
      instruction 200
single-precision floating-point
  adding 32-bit operand to result of multiplying two
    operands
    using fmadds (Floating Multiply-Add Single)
      instruction 216, 236
  adding two 32-bit operands
    using fadds (Floating Add Single) instruction 203
  dividing 32-bit operands
    using fdivs (Floating Divide Single)
      instruction 213
  multiplying two 32-bit operands
    using fmuls (Floating Multiply Single)
      instruction 221
  multiplying two 32-bit operands and adding to 32-bit
    operand
    using fnmadds (Floating Negative Multiply-Add Single) instruction 226
  multiplying two 32-bit operands and subtracting 32-bit
    operand
    using fnmsubs (Floating Negative Multiply-Subtract Single) instruction 228
  subtracting 32-bit operand from result of multiplying
    two 32-bit operands
    using fnmsubs (Floating Multiply-Subtract Single)
      instruction 219
  subtracting 32-bit operands
    using fsubs (Floating Subtract Single)
      instruction 240
sl (Shift Left) instruction 373
sld (Shift Left Double Word) Instruction 364
sldi extended mnemonic 112
sl (Shift Left Extended) instruction 364
sle (Shift Left Extended with MQ) instruction 366
sqliq (Shift Left Immediate with MQ) instruction 367
slliq (Shift Left Long Immediate with MQ) instruction 369
slq (Shift Left with MQ) instruction 370
slw (Shift Left Word) instruction 373
slwi extended mnemonic 109
source files
identifying file names using .file pseudo-op 482
source language type 5
identifying using .source pseudo-op 498
source module
identifying a symbol defined in another using .extern pseudo-op 481
special-purpose registers
changes and field handling 8
copying general-purpose register contents into using mtspr (Move to Special-Purpose Register) instruction 315
copying the contents into a general-purpose register using mfspr (Move from Special-Purpose Register) instruction 303
extended mnemonics 102
split-field notation 16
square root, reciprocal floating estimate 235
sr (Shift Right) instruction 395
sra (Shift Right Algebraic) instruction 380
srad (Shift Right Algebraic Double Word) instruction 375
sradi (Shift Right Algebraic Double Word Immediate) instruction 376
srai (Shift Right Algebraic Immediate) instruction 382
sraq (Shift Right Algebraic with MQ) instruction 377
sraw (Shift Right Algebraic Word) instruction 380
srawi (Shift Right Algebraic Word Immediate) instruction 382
srd (Shift Right Double Word) instruction 383
srbi extended mnemonic 112
sre (Shift Right Extended) instruction 384
srea (Shift Right Extended Algebraic) instruction 386
sreq (Shift Right Extended with MQ) instruction 387
sriq (Shift Right Immediate with MQ) instruction 389
srlq (Shift Right Long Immediate with MQ) instruction 390
srlq (Shift Right Long with MQ) instruction 392
srq (Shift Right with MQ) instruction 393
srw (Shift Right Word) instruction 395
srwi extended mnemonic 109
st (Store) instruction 428
stack
runtime process 67
stack-related system standards 70
statements 29
static blocks
identifying the beginning of using .bs pseudo-op 469
static blocks (continued)
identifying the end of using .es pseudo-op 481
static name
keeping information in the symbol table using .globl pseudo-op 487
stb (Store Byte) instruction 396
stbrx (Store Byte-Reverse Indexed) instruction 429
stbu (Store Byte with Update) instruction 397
stbux (Store Byte with Update Indexed) instruction 398
stbx (Store Byte Indexed) instruction 399
std (Store Double Word) Instruction 400
stdcx. (Store Double Word Conditional Indexed) Instruction 401
stdu (Store Double Word with Update) Instruction 402
stdux (Store Double Word with Update Indexed) Instruction 403
stdx (Store Double Word Indexed) Instruction 404
stfd (Store Floating-Point Double) instruction 405
stfdcx (Store Floating-Point Double with Update) instruction 406
stfdux (Store Floating-Point Double with Update Indexed) instruction 407
stfdx (Store Floating-Point Double Indexed) instruction 408
stfw (Store Floating-Point as Integer Word Indexed) instruction 409
stfq (Store Floating-Point Quad) instruction 410
stfqu (Store Floating-Point Quad with Update) instruction 411
stfqux (Store Floating-Point Quad with Update Indexed) instruction 412
stfq (Store Floating-Point Quad Indexed) instruction 413
stfs (Store Floating-Point Single) instruction 414
stfsu (Store Floating-Point Single with Update) instruction 415
stfdux (Store Floating-Point Single with Update Indexed) instruction 416
stfsx (Store Floating-Point Single Indexed) instruction 417
sth (Store Half) instruction 418
stbrx (Store Half Byte-Reverse Indexed) instruction 419
sthu (Store Half with Update) instruction 420
sthux (Store Half with Update Indexed) instruction 421
sthx (Store Half Indexed) instruction 422
stm (Store Multiple) instruction 423
stmw (Store Multiple Word) instruction 423
storage
synchronize using sync (Synchronize) instruction 447
storage definition
pseudo-ops 464
storage mapping classes 473
store
quad word 424
stq (Store Quad Word) instruction 424
stoi (Store String Immediate) instruction 425
stswi (Store String Word Immediate) instruction 425
stswx (Store String Word Indexed) instruction 426
stx (Store Indexed) instruction 434
stu (Store with Update) instruction 431
stux (Store with Update Indexed) instruction 433
stw (Store Word) instruction 428
stwbx (Store Word Byte-Reverse Indexed) instruction 429
stwcx. (Store Word Conditional Indexed) instruction 430
stwu (Store Word with Update) instruction 431
stwux (Store Word with Update Indexed) instruction 433
stwx (Store Word Indexed) instruction 434
stx (Store Indexed) instruction 434
sub[][.] extended mnemonic 97
subc[][.] extended mnemonic 97
subf (Subtract From) instruction 435
subfc (Subtract from Carrying) instruction 437
subfe (Subtract from Extended) instruction 439
subfic (Subtract from Immediate Carrying) instruction 441
subfme (Subtract from Minus One Extended) instruction 442
subfze (Subtract from Zero Extended) instruction 444
subi extended mnemonic 97
subif (Subtract Immediate from) instruction 436
subic[][.] extended mnemonic 97
subis extended mnemonic 97
subroutine
  linkage convention 65
svc (Supervisor Call) instruction 446
symbol table
  entries for debuggers
    pseudo-ops 465
  keeping information of a static name in the using .globl pseudo-op 487
symbols
  associating a hash value with external
    using .hash pseudo-op 485
  constructing 31
  interpreting a cross-reference 63
  making globally visible to linker
    using .globl pseudo-op 484
  setting equal to an expression in type and value
    using .set pseudo-op 496
sync (Synchronize) instruction 447
synchronize
  using isync (Instruction Synchronize) instruction 243
syntax and semantics
  character set 27
  comments 31
  constants 35
  constructing symbols 31
  expressions 39
  instruction statements 29
  labels 30
  line format 28
  mnemonics 30
  null statements 29
  operands 31
  operators 38
  pseudo-operation statements 29
  reserved words 28

T

t (Trap) instruction 456
table of contents
  defining
    using .toc pseudo-op 504
tags
  traceback 78
target addresses
  branching conditionally to
    using bc (Branch Conditional) instruction 144
  branching to
    using b (Branch) instruction 143
target environment
  defining
    using .machine pseudo-op 490
  indicator flag 4
td (Trap Double Word) Instruction 449
tdi (Trap Double Word Immediate) Instruction 450
ti (Trap Immediate) Instruction 457
tbli (Translation Look-Aside Buffer Invalidate Entry) Instruction 451
tbie (Translation Look-Aside Buffer Invalidate Entry) Instruction 451
tbld (Load Data TLB Entry) instruction 452
tblli (Load Instruction TLB Entry) instruction 454
tlbsync (Translation Look-Aside Buffer Synchronize) Instruction 455
TOC
  accessing data through 83
  intermodule calls using 85
  programming the 82
  understanding the 82
  traceback tags 78
tw (Trap Word) instruction 456
twi (Trap Word Immediate) instruction 457

U

user register set
  POWER family 12
  PowerPC 12
  using .weak pseudo-op 510

V

Vector Processor 597
depth stabstrings 603
legacy ABI
  compatibility 604
  interoperability 604
procedure calling sequence 601
  argument passing 601
  function return values 603
register usage conventions 597
run-time stack 598
storage operands and alignment 597
Vector Processor (continued)
  traceback tables 603
  vector register
    save and restore 600

W
  warning messages 8, 513

X
  xor (XOR) instruction 458
  xori (XOR Immediate) instruction 459
  xoril (XOR Immediate Lower) instruction 459
  xoris (XOR Immediate Shift) instruction 460
  xoriu (XOR Immediate Upper) instruction 460
Readers’ Comments — We’d Like to Hear from You

AIX 5L Version 5.3
Assembler Language Reference

Publication No. SC23-4923-02

Overall, how satisfied are you with the information in this book?

<table>
<thead>
<tr>
<th>Overall satisfaction</th>
<th>Very Satisfied</th>
<th>Satisfied</th>
<th>Neutral</th>
<th>Dissatisfied</th>
<th>Very Dissatisfied</th>
</tr>
</thead>
</table>

How satisfied are you that the information in this book is:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Very Satisfied</th>
<th>Satisfied</th>
<th>Neutral</th>
<th>Dissatisfied</th>
<th>Very Dissatisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accurate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easy to find</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easy to understand</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Well organized</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicable to your tasks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please tell us how we can improve this book:

Thank you for your responses. May we contact you?  ☐ Yes  ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Name

Address

Company or Organization

Phone No.