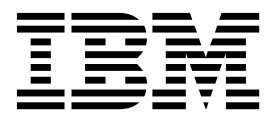


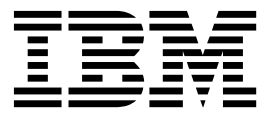
AIX Version 6.1

Assembler Language Reference



AIX Version 6.1

Assembler Language Reference



Note

Before using this information and the product it supports, read the information in “Notices” on page 643.

This edition applies to AIX Version 6.1 and to all subsequent releases and modifications until otherwise indicated in new editions.

© **Copyright IBM Corporation 1997, 2018.**

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

Contents

About this document ix

Highlighting ix

Case sensitivity in AIX ix

ISO 9000 ix

Assembler language reference 1

What's new in Assembler Language Reference 1

Assembler overview 1

 Features of the AIX assembler 1

 Assembler installation 9

Processing and storage 9

 POWER family and PowerPC architecture

 overview 10

 Branch processor 19

 Fixed-point processor 21

 Floating-point processor 28

Syntax and semantics 32

 Character set 32

 Reserved words 33

 Line format 34

 Statements 35

 Symbols 39

 Relocation specifiers 47

 Constants 47

 Operators 55

 Expressions 57

Addressing 68

 Absolute addressing 68

 Absolute immediate addressing 68

 Relative immediate addressing 69

 Explicit-based addressing 70

 Implicit-based addressing 71

 Location counter 72

Assembling and linking a program 73

 Assembling and linking a program 73

 Understanding assembler passes 75

 Interpreting an assembler listing 77

 Interpreting a symbol cross-reference 82

 Subroutine linkage convention 83

 Understanding and programming the toc 110

 Using thread local storage 116

 Running a program 118

Extended instruction mnemonics 118

 Extended mnemonics of branch instructions 119

 Extended mnemonics of condition register

 logical instructions 126

 Extended mnemonics of fixed-point arithmetic

 instructions 127

 Extended mnemonics of fixed-point compare

 instructions 128

 Extended mnemonics of fixed-point load

 instructions 129

 Extended mnemonics of fixed-point logical

 instructions 130

 Extended mnemonics of fixed-point trap

 instructions 131

 Extended mnemonic mctr for moving to the

 condition register 132

 Extended mnemonics of moving from or to

 special-purpose registers 132

 Extended mnemonics of 32-bit fixed-point rotate

 and shift instructions 137

 Extended mnemonics of 64-bit fixed-point rotate

 and shift instructions 140

Migrating source programs 143

 Functional differences for POWER family and

 PowerPC instructions 144

 Differences between POWER family and

 PowerPC instructions with the same op code 145

 Extended mnemonics changes 147

 POWER family instructions deleted from

 PowerPC 151

 Added PowerPC instructions 152

 Instructions available only for the PowerPC 601

 RISC microprocessor 152

 Migration of branch conditional statements with

 no separator after mnemonic 153

Instruction set 154

 abs (Absolute) instruction 154

 add (Add) or cax (Compute Address)

 instruction 156

 addc or a (Add Carrying) instruction 158

 adde or ae (Add Extended) instruction 160

 addi (Add Immediate) or cal (Compute Address

 Lower) instruction 162

 addic or ai (Add Immediate Carrying)

 instruction 163

 addic. or ai. (Add Immediate Carrying and

 Record) instruction 164

 addis or cau (Add Immediate Shifted)

 instruction 165

 addme or ame (Add to Minus One Extended)

 instruction 167

 addze or aze (Add to Zero Extended)

 instruction 169

 and (AND) instruction 171

 andc (AND with Complement) instruction 172

 andi. or andil. (AND Immediate) instruction 174

 andis. or andiu. (AND Immediate Shifted)

 instruction 175

 b (Branch) instruction 176

 bc (Branch Conditional) instruction 177

 bcctr or bcc (Branch Conditional to Count

 Register) instruction 179

 bclr or bcr (Branch Conditional Link Register)

 instruction 182

 clcs (Cache Line Compute Size) instruction 184

 clf (Cache Line Flush) instruction 186

 cli (Cache Line Invalidate) instruction 187

 cmp (Compare) instruction 189

cmpi (Compare Immediate) instruction	191	fdiv or fd (Floating Divide) instruction	251
cmpl (Compare Logical) instruction	192	fmadd or fma (Floating Multiply-Add)	
cmpli (Compare Logical Immediate) instruction	193	instruction	253
cntlzd (Count Leading Zeros Double Word)		fmr (Floating Move Register) instruction	256
instruction	195	fmsub or fms (Floating Multiply-Subtract)	
cntlzw or cntlz (Count Leading Zeros Word)		instruction	257
instruction	196	fmul or fm (Floating Multiply) instruction	259
crand (Condition Register AND) instruction	197	fnabs (Floating Negative Absolute Value)	
crandc (Condition Register AND with		instruction	262
Complement) instruction	198	fneg (Floating Negate) instruction	263
creqv (Condition Register Equivalent)		fnmadd or fnma (Floating Negative	
instruction	199	Multiply-Add) instruction	264
crnand (Condition Register NAND) instruction	200	fnmsub or fnms (Floating Negative	
crnor (Condition Register NOR) instruction	201	Multiply-Subtract) instruction	267
cror (Condition Register OR) instruction	202	fres (Floating Reciprocal Estimate Single)	
crorc (Condition Register OR with Complement)		instruction	269
instruction	203	frsp (Floating Round to Single Precision)	
crxor (Condition Register XOR) instruction	204	instruction	271
dcbf (Data Cache Block Flush) instruction	205	frsqrt (Floating Reciprocal Square Root	
dcbi (Data Cache Block Invalidate) instruction	206	Estimate) instruction	273
dcbst (Data Cache Block Store) instruction	208	fsel (Floating-Point Select) instruction	275
dcbt (Data Cache Block Touch) instruction	209	fsqrt (Floating Square Root, Double-Precision)	
dcbtst (Data Cache Block Touch for Store)		instruction	276
instruction	212	fsqrts (Floating Square Root Single) instruction	278
dcbz or dclz (Data Cache Block Set to Zero)		fsb or fs (Floating Subtract) instruction	279
instruction	214	icbi (Instruction Cache Block Invalidate)	
dclst (Data Cache Line Store) instruction	216	instruction	282
div (Divide) instruction	217	isync or ics (Instruction Synchronize) instruction	283
divd (Divide Double Word) instruction	219	lbz (Load Byte and Zero) instruction	284
divdu (Divide Double Word Unsigned)		lbzu (Load Byte and Zero with Update)	
instruction	220	instruction	285
divs (Divide Short) instruction	222	lbzux (Load Byte and Zero with Update	
divw (Divide Word) instruction	224	Indexed) instruction	286
divwu (Divide Word Unsigned) instruction	225	lbzx (Load Byte and Zero Indexed) instruction	288
doz (Difference or Zero) instruction	227	ld (Load Doubleword) instruction	289
dozi (Difference or Zero Immediate) instruction	229	ldarx (Load Doubleword Reserve Indexed)	
eciwx (External Control In Word Indexed)		instruction	290
instruction	230	ldu (Load Doubleword with Update) instruction	291
ecowx (External Control Out Word Indexed)		ldux (Load Doubleword with Update Indexed)	
instruction	231	instruction	293
eiio (Enforce In-Order Execution of I/O)		ldx (Load Doubleword Indexed) instruction	293
instruction	232	lfd (Load Floating-Point Double) instruction	294
extsw (Extend Sign Word) instruction	233	lfdx (Load Floating-Point Double with Update)	
eqv (Equivalent) instruction	234	instruction	295
extsb (Extend Sign Byte) instruction	236	lfdx (Load Floating-Point Double with Update	
extsh or exts (Extend Sign Halfword) instruction	237	Indexed) instruction	296
fabs (Floating Absolute Value) instruction	238	lfdx (Load Floating-Point Double-Indexed)	
fadd or fa (Floating Add) instruction	240	instruction	298
fcfid (Floating Convert from Integer Double		lfq (Load Floating-Point Quad) instruction	299
Word) instruction	242	lfqu (Load Floating-Point Quad with Update)	
fcmpo (Floating Compare Ordered) instruction	243	instruction	300
fcmpu (Floating Compare Unordered)		lfqux (Load Floating-Point Quad with Update	
instruction	244	Indexed) instruction	301
ftid (Floating Convert to Integer Double Word)		lfqx (Load Floating-Point Quad Indexed)	
instruction	245	instruction	303
ftidz (Floating Convert to Integer Double Word		lfs (Load Floating-Point Single) instruction	304
with Round toward Zero) instruction	246	lfsu (Load Floating-Point Single with Update)	
ftiw or fcir (Floating Convert to Integer Word)		instruction	305
instruction	247	lfsux (Load Floating-Point Single with Update	
ftiwz or fcirz (Floating Convert to Integer		Indexed) instruction	306
Word with Round to Zero) instruction	249		

lfsx (Load Floating-Point Single Indexed)	
instruction	307
lha (Load Half Algebraic) instruction	308
lhau (Load Half Algebraic with Update)	
instruction	309
lhaux (Load Half Algebraic with Update	
Indexed) instruction	310
lhax (Load Half Algebraic Indexed) instruction	311
lhrx (Load Half Byte-Reverse Indexed)	
instruction	312
lh (Load Half and Zero) instruction	314
lhzu (Load Half and Zero with Update)	
instruction	314
lhzu (Load Half and Zero with Update	
Indexed) instruction	316
lhzu (Load Half and Zero Indexed) instruction	317
lmw or lm (Load Multiple Word) instruction	318
lq (Load Quad Word) instruction	319
lscbx (Load String and Compare Byte Indexed)	
instruction	320
lswi or lsi (Load String Word Immediate)	
instruction	322
lswx or lsx (Load String Word Indexed)	
instruction	324
lwa (Load Word Algebraic) instruction	326
lwarx (Load Word and Reserve Indexed)	
instruction	327
lwaux (Load Word Algebraic with Update	
Indexed) instruction	328
lwax (Load Word Algebraic Indexed) instruction	329
lwbx or lbrx (Load Word Byte-Reverse	
Indexed) instruction	330
lwz or l (Load Word and Zero) instruction . .	331
lwzu or lu (Load Word with Zero Update)	
instruction	332
lwzu or lux (Load Word and Zero with Update	
Indexed) instruction	333
lwzu or lx (Load Word and Zero Indexed)	
instruction	334
maskg (Mask Generate) instruction	336
maskr (Mask Insert from Register) instruction	337
mcrf (Move Condition Register Field)	
instruction	339
mcrfs (Move to Condition Register from FPSCR)	
instruction	339
mcrxr (Move to Condition Register from XER)	
instruction	341
mfr (Move from Condition Register) instruction	342
mffs (Move from FPSCR) instruction	342
mfmsr (Move from Machine State Register)	
instruction	344
mfocrf (Move from One Condition Register	
Field) instruction	345
mfsp (Move from Special-Purpose Register)	
instruction	346
mfsr (Move from Segment Register) instruction	348
mfstri (Move from Segment Register Indirect)	
instruction	349
mfstri (Move from Segment Register Indirect)	
instruction	350

mtrcf (Move to Condition Register Fields)	
instruction	351
mtfsb0 (Move to FPSCR Bit 0) instruction . . .	352
mtfsb1 (Move to FPSCR Bit 1) instruction . . .	354
mtfsf (Move to FPSCR Fields) instruction . . .	355
mtfsfi (Move to FPSCR Field Immediate)	
instruction	357
mtocrf (Move to One Condition Register Field)	
instruction	358
mtspr (Move to Special-Purpose Register)	
instruction	359
mul (Multiply) instruction	361
mulhd (Multiply High Double Word) instruction	363
mulhdu (Multiply High Double Word	
Unsigned) instruction	364
mulhw (Multiply High Word) instruction . . .	365
mulhwu (Multiply High Word Unsigned)	
instruction	367
mulld (Multiply Low Double Word) instruction	368
mulli or muli (Multiply Low Immediate)	
instruction	369
mullw or muls (Multiply Low Word) instruction	370
nabs (Negative Absolute) instruction	372
nand (NAND) instruction	374
neg (Negate) instruction	375
nor (NOR) instruction	377
or (OR) instruction	378
orc (OR with Complement) instruction	380
ori or oril (OR Immediate) instruction	381
oris or oriu (OR Immediate Shifted) instruction	382
popcntbd (Population Count Byte Doubleword)	
instruction	383
rac (Real Address Compute) instruction . . .	384
rfi (Return from Interrupt) instruction	386
rfd (Return from Interrupt Double Word)	
instruction	386
rfsvc (Return from SVC) instruction	387
rldcl (Rotate Left Double Word then Clear Left)	
instruction	388
rldicl (Rotate Left Double Word Immediate then	
Clear Left) instruction	389
rldcr (Rotate Left Double Word then Clear	
Right) instruction	390
rldic (Rotate Left Double Word Immediate then	
Clear) instruction	391
rldicl (Rotate Left Double Word Immediate then	
Clear Left) instruction	392
rldicr (Rotate Left Double Word Immediate then	
Clear Right) instruction	394
rldimi (Rotate Left Double Word Immediate	
then Mask Insert) instruction	395
rlmi (Rotate Left Then Mask Insert) instruction	396
rlwimi or rlimi (Rotate Left Word Immediate	
Then Mask Insert) instruction	398
rlwinm or rlinm (Rotate Left Word Immediate	
Then AND with Mask) instruction	399
rlwnm or rlnm (Rotate Left Word Then AND	
with Mask) instruction	401
rrib (Rotate Right and Insert Bit) instruction . .	404
sc (System Call) instruction	405
scv (System Call Vectored) instruction	406

si (Subtract Immediate) instruction	407
si. (Subtract Immediate and Record) instruction	408
sld (Shift Left Double Word) instruction	409
sle (Shift Left Extended) instruction	410
sleq (Shift Left Extended with MQ) instruction	412
sliq (Shift Left Immediate with MQ) instruction	413
slliq (Shift Left Long Immediate with MQ) instruction	415
sllq (Shift Left Long with MQ) instruction	416
slq (Shift Left with MQ) instruction	418
slw or sl (Shift Left Word) instruction	419
srad (Shift Right Algebraic Double Word) instruction	421
sradi (Shift Right Algebraic Double Word Immediate) instruction	422
sraiq (Shift Right Algebraic Immediate with MQ) instruction	423
sraq (Shift Right Algebraic with MQ) instruction	425
sraw or sra (Shift Right Algebraic Word) instruction	427
srawi or srai (Shift Right Algebraic Word Immediate) instruction	428
srđ (Shift Right Double Word) instruction	430
sre (Shift Right Extended) instruction	431
srea (Shift Right Extended Algebraic) instruction	432
sreq (Shift Right Extended with MQ) instruction	434
sriq (Shift Right Immediate with MQ) instruction	436
srlq (Shift Right Long Immediate with MQ) instruction	437
srlq (Shift Right Long with MQ) instruction	439
srq (Shift Right with MQ) instruction	440
srw or sr (Shift Right Word) instruction	442
stb (Store Byte) instruction	443
stbu (Store Byte with Update) instruction	444
stbux (Store Byte with Update Indexed) instruction	445
stbx (Store Byte Indexed) instruction	447
std (Store Double Word) instruction	448
stdcx. (Store Double Word Conditional Indexed) instruction	448
stdu (Store Double Word with Update) instruction	450
stdux (Store Double Word with Update Indexed) instruction	451
stdx (Store Double Word Indexed) instruction	452
stfd (Store Floating-Point Double) instruction	453
stfdu (Store Floating-Point Double with Update) instruction	454
stfdx (Store Floating-Point Double with Update Indexed) instruction	455
stfdx (Store Floating-Point Double Indexed) instruction	456
stfiwx (Store Floating-Point as Integer Word indexed)	457
stfq (Store Floating-Point Quad) instruction	458
stfqu (Store Floating-Point Quad with Update) instruction	459
stfqx (Store Floating-Point Quad with Update Indexed) instruction	460

stfqx (Store Floating-Point Quad Indexed) instruction	461
stfs (Store Floating-Point Single) instruction	462
stfsu (Store Floating-Point Single with Update) instruction	463
stfsux (Store Floating-Point Single with Update Indexed) instruction	464
stfsx (Store Floating-Point Single Indexed) instruction	466
sth (Store Half) instruction	467
sthrx (Store Half Byte-Reverse Indexed) instruction	467
sthu (Store Half with Update) instruction	469
sthux (Store Half with Update Indexed) instruction	470
sthx (Store Half Indexed) instruction	471
stmw or stm (Store Multiple Word) instruction	472
stq (Store Quad Word) instruction	473
stswi or stsi (Store String Word Immediate) instruction	474
stswx or stsx (Store String Word Indexed) instruction	476
stw or st (Store) instruction	477
stwbrx or stbrx (Store Word Byte-Reverse Indexed) instruction	478
stwcx. (Store Word Conditional Indexed) instruction	480
stwu or stu (Store Word with Update) instruction	481
stwux or stux (Store Word with Update Indexed) instruction	483
stwx or stx (Store Word Indexed) instruction	484
subf (Subtract From) instruction	485
subfc or sf (Subtract from Carrying) instruction	487
subfe or sfe (Subtract from Extended) instruction	489
subfic or sfi (Subtract from Immediate Carrying) instruction	491
subfme or sfme (Subtract from Minus One Extended) instruction	492
subfze or sfze (Subtract from Zero Extended) instruction	494
svc (Supervisor Call) instruction	496
sync (Synchronize) or dcs (Data Cache Synchronize) instruction	498
td (Trap Double Word) instruction	499
tdi (Trap Double Word Immediate) instruction	501
tlbie or tlbi (Translation Look-Aside Buffer Invalidate Entry) instruction	502
tlbld (Load Data TLB Entry) instruction	503
tlbli (Load Instruction TLB Entry) instruction	505
tlbli instruction function	506
tlbsync (Translation Look-Aside Buffer Synchronize) instruction	506
tw or t (Trap Word) instruction	507
twi or ti (Trap Word Immediate) instruction	508
xor (XOR) instruction	509
xori or xoril (XOR Immediate) instruction	510
xoris or xoriu (XOR Immediate Shift) instruction	511
Pseudo-ops	512
Pseudo-ops overview	512

.align pseudo-op	517	.space pseudo-op	554
.bb pseudo-op	518	.stabx pseudo-op	555
.bc pseudo-op	518	.string pseudo-op	556
.bf pseudo-op	519	.tbttag pseudo-op	556
.bi pseudo-op	520	.tc pseudo-op	558
.bs pseudo-op	520	.toc pseudo-op	559
.byte pseudo-op	521	.tocof pseudo-op	560
.comm pseudo-op	522	.using pseudo-op	561
.comment pseudo-op	524	.vbyte pseudo-op	565
.csect pseudo-op	525	.weak pseudo-op	566
.double pseudo-op	527	.xline pseudo-op	567
.drop pseudo-op	528	Appendix A messages	568
.dsect pseudo-op	529	Appendix B instruction set sorted by mnemonic	585
.eb pseudo-op	531	Appendix C instruction set sorted by primary and extended op code	595
.ec pseudo-op	532	Appendix D instructions common to POWER family, POWER2, and PowerPC	606
.ef pseudo-op	532	Appendix E POWER family and POWER2 instructions	609
.ei pseudo-op	533	Appendix F PowerPC instructions	616
.es pseudo-op	534	Appendix G PowerPC 601 RISC Microprocessor instructions	624
.except pseudo-op	534	Appendix H value definitions	632
.extern pseudo-op	535	Appendix I vector processor	634
.file pseudo-op	536	Storage operands and alignment	634
.float pseudo-op	537	Register usage conventions	634
.function pseudo-op	537	Runtime stack	635
.globl pseudo-op	538	Procedure calling sequence	638
.hash pseudo-op	539	Traceback tables	640
.info pseudo-op	540	Debug stabstrings	641
.lcomm pseudo-op	541	Legacy ABI compatibility and interoperability	641
.lglobl pseudo-op	542		
.line pseudo-op	543		
.long pseudo-op	544		
.llong pseudo-op	544		
.machine pseudo-op	545		
.org pseudo-op	548		
.quad pseudo-op	549		
.ref pseudo-op	549		
.rename pseudo-op	550		
.set pseudo-op	551		
.short pseudo-op	552		
.source pseudo-op	553		
		Notices	643
		Privacy policy considerations	645
		Trademarks	645
		Index	647

About this document

This document provides users with detailed information about the assembler program that operates within the operating system. This topic collection also contains details about pseudo-ops and instruction sets. This publication is also available on the documentation CD that is shipped with the operating system.

Highlighting

The following highlighting conventions are used in this document:

Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Bold highlighting also identifies graphical objects, such as buttons, labels, and icons that the you select.
<i>Italics</i>	Identifies parameters for actual names or values that you supply.
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 text that you must type.

Case sensitivity in AIX

Everything in the AIX[®] 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.

Assembler language reference

The **Assembler Language Reference** topic provides information about the assembler program that operates within the operating system.

The assembler takes machine-language instructions and translates them into machine object code.

What's new in Assembler Language Reference

Read about new or significantly changed information for the Assembler Language Reference topic collection.

How to see what's new or changed

In this PDF file, you might see revision bars (|) in the left margin that identifies new and changed information.

March 2018

Added information about support for POWER9™ processor-based servers in the following topics:

- Multiple hardware architecture and implementation platform support
- “CPU ID definition” on page 4

October 2017

Added information about following pseudo-ops:

- “.comment pseudo-op” on page 524
- “.except pseudo-op” on page 534
- “.info pseudo-op” on page 540

Updated information about following pseudo-ops:

- “.file pseudo-op” on page 536
- “Miscellaneous” on page 515

Assembler overview

The assembler program takes machine-language instructions and translates them into machine object-code.

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

Features of AIX Assembler.

This section describes features of the AIX® assembler.

Related concepts:

“Assembler overview”

The assembler program takes machine-language instructions and translates them into machine

object-code.

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

Related information:

as

Multiple hardware architecture and implementation platform support

The assembler supports source programs containing instructions unique to any of the Power® and PowerPC® processor architectures.

The following Power and PowerPC processor architectures are supported:

- The first-generation POWER® family processors (POWER family architecture)
- The POWER2 processors (POWER family architecture)
- The PowerPC 601 RISC Microprocessor, PowerPC 604 RISC Microprocessor, or the PowerPC A35 RISC Microprocessor (PowerPC architecture)
- The POWER4, POWER5, PowerPC 970, POWER5+, POWER6®, POWER7®, POWER8®, and POWER9 processors

There are several categories of instructions, and one or more categories of instructions are valid on each supported implementation. The various architecture descriptions describe the supported instructions for each implementation. The **-M** flag can be used to determine which instructions are valid in a particular assembly mode or to determine which assembly modes allow a certain instruction to be used.

- | The POWER9 processor architecture is described in the Power Instruction Set Architecture Version 3.0 specification. For more information, see the OpenPOWER website.

Related concepts:

“Assembler overview” on page 1

The assembler program takes machine-language instructions and translates them into machine object-code.

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

Related information:

as

Host machine independence and target environment indicator flag

The host machine is the hardware platform on which the assembler runs.

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 chosen 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.

Related concepts:

“Assembler overview” on page 1

The assembler program takes machine-language instructions and translates them into machine object-code.

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“.machine pseudo-op” on page 545

Related information:

as

Mnemonics cross-reference

The assembler supports both PowerPC® and POWER® family mnemonics.

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.

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.

Related concepts:

“Assembler overview” on page 1

The assembler program takes machine-language instructions and translates them into machine object-code.

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

Related information:

as

CPU ID definition

During the assembly process the assembler determines the smallest instruction set containing all the instructions and gives a CPU ID value indicating the instruction set.

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:

Value	Description
com	All instructions used in the program are in the PowerPC® and POWER® family architecture intersection. (The com instruction set is the smallest instruction set.)
ppc	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.)
pwr	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.)
pwr2	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.)
any	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.)

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:

Low-Order Byte	CPU ID
0	Not a defined value. An invalid value or object was assembled prior to definition of the CPU-ID field.
1	ppc
2	ppc64
3	com
4	pwr
5	any
6	601
7	603
8	604
10	power
16	620
17	A35
18	pwr5
19	ppc970 or 970
20	pwr6
21	vec

Low-Order Byte	CPU ID
22	pwr5x
23	pwr6e
24	pwr7
25	pwr8
26	pwr9
224	pwr2 or pwrx

Related concepts:

“Assembler overview” on page 1

The assembler program takes machine-language instructions and translates them into machine object-code.

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

Related information:

as

Source language type

The assembler records the 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:

High-Order Byte	Language
0x00	C
0x01	FORTRAN
0x02	Pascal
0x03	Ada
0x04	PL/I
0x05	Basic
0x06	Lisp
0x07	Cobol
0x08	Modula2
0x09	C++
0x0A	RPG
0x0B	PL8, PLIX
0x0C	Assembler
0x0D-BxFF	Reserved

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.

Related concepts:

“Assembler overview” on page 1

The assembler program takes machine-language instructions and translates them into machine object-code.

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

Related information:

as

Detection error conditions

An error is reported if the source program contains invalid instruction forms. Error occurs due to incompatibility between the POWER[®] family and PowerPC[®] architectures.

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 / (slash) (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 **svca** 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[®] **mtsriin** 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:

BO	Description
0000y	Decrement the Count Register (CTR); then branch if the value of the decremented CTR is not equal to 0 and the condition is False.
0001y	Decrement the CTR; then branch if the value of the decremented CTR is not equal to 0 and the condition is False.
001zy	Branch if the condition is False.
0100y	Decrement the CTR; then branch if the value of the decremented CTR is not equal to 0 and the condition is True.
0101y	Decrement the CTR; then branch if the value of the decremented CTR is not equal to 0 and the condition is True.
011zy	Branch if the condition is True.
1z00y	Decrement the CTR; then branch if the value of the decremented CTR is not equal to 0.
1z01y	Decrement the CTR; then branch if the value of the decremented CTR is not equal to 0.
1z1zz	Branch always.

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.
 - **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 run time.

- 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**
 - **cmpi**
 - **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

The warning messages are listed when the **-w** flag is used with the **as** command.

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 1. Same Op Codes with Different Mnemonics

POWER® family	PowerPC®
dcs	sync
ics	isync
svca	sc
mtsri	mtsrin
lsx	lswx

- 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**

Related concepts:

“Assembler overview” on page 1

The assembler program takes machine-language instructions and translates them into machine object-code.

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

Related information:

as

Special-purpose register changes and special-purpose register field handling

The special-purpose registers are defined in the POWER® family architecture.

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, DCMR, 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 **mtspr** and **mfscr** 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 **mfscr** or **mtspr** instruction.

Related information:

as

Assembler installation

The AIX® assembler is installed with the base operating system, along with the commands, files, and libraries.

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

Related concepts:

“.machine pseudo-op” on page 545

“.source pseudo-op” on page 553

Related information:

as

Processing and storage

The processor stores the data in the main memory and in the registers.

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 topic 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 topic are nonprivileged. Therefore, all the registers discussed in this topic 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

A POWER[®] family or PowerPC[®] microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

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.

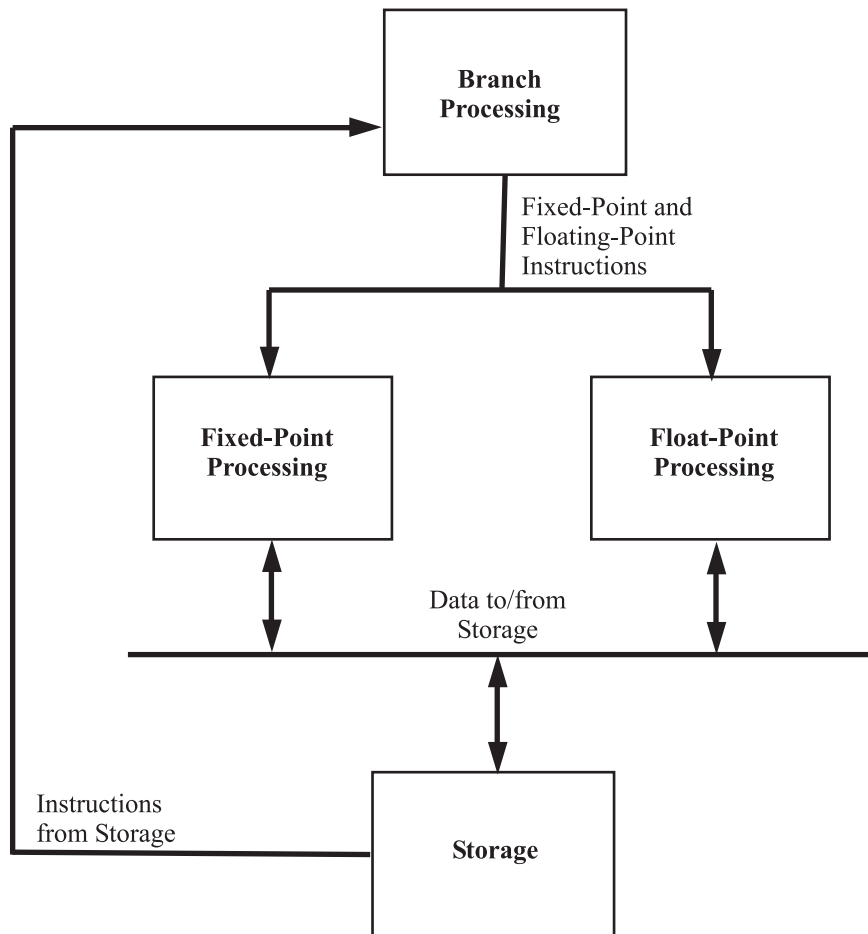


Figure 1. Logical Processing Model. The process begins at the top with Branch Processing, which branches to either fixed-point or float-point processing. These processes send and receive data from storage. Storage will also send more instructions to Branch Processing at the top of the diagram.

The following table shows the registers for 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.

Register	Bits Available
Condition Register (CR)	0-31
Link Register (LR)	0-31
Count Register (CTR)	0-31
General Purpose Registers 00-31 (GPR)	0-31 for each register
Fixed-Point Exception Register (XER)	0-31
Floating-Point Registers 00-31 (FPR)	0-63 for each register
Floating Point Status and Control Register (FPSCR)	0-31

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.

Register	Bits Available
Condition Register (CR)	0-31
Link Register (LR)	0-31
Count Register (CTR)	0-31
General Purpose Registers 00-31 (GPR)	0-31 for each register
Multiply-Quotient Register (MQ)	0-31
Fixed-Point Exception Register (XER)	0-31
Floating-Point Registers 00-31 (FPR)	0-63 for each register
Floating Point Status and Control Register (FPSCR)	0-31

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.

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the `hello.s` assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Instruction forms

The instructions are four bytes long and are word-aligned.

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**

Bits	Value
0-5	OPCD
6-29	LI
30	AA
31	LK

- **B Form**

Bits	Value
0-5	OPCD
6-10	BO
11-15	BI
16-29	BD
30	AA
31	LK

- **SC Form**

Bits	Value
0-5	OPCD
6-10	///
11-15	///
16-29	///
30	XO
31	/

- **D Form**

Bits	Value
0-5	OPCD
6-10	RT, RS, FRT, FRS, TO, or BF, /, and L
11-15	RA
16-31	D, SI, or UI

- **DS Form**

Bits	Value
0-5	OPCD
6-10	RT or RS
11-15	RA
16-29	DS
30-31	XO

- **X Instruction Format**

Bits	Value
0-5	OPCD
6-10	RT, FRT, RS, FRS, TO, BT, or BE, /, and L
11-15	RA, FRA, SR, SPR, or BFA and //
16-20	RB, FRB, SH, NB, or U and /
21-30	XO or EO
31	Rc

– **XL Instruction Format**

Bits	Value
0-5	OPCD
6-10	RT or RS
11-20	spr or /, FXM and /
21-30	XO or EO
31	Rc

– **XFX Instruction Format**

Bits	Value
0-5	OPCD
6-10	RT or RS
11-20	spr or /, FXM and /
21-30	XO or EO
31	Rc

– **XFL Instruction Format**

Bits	Value
0-5	OPCD
6	/
7-14	FLM
15	/
16-20	FRB
21-30	XO or EO
31	Rc

– **XO Instruction Format**

Bits	Value
0-5	OPCD
6-10	RT
11-15	RA
16-20	RB
21	OE
22-30	XO or EO
31	Rc

• **A Form**

Bits	Value
0-5	OPCD
6-10	FRT
11-15	FRA
16-20	FRB
21-25	FRC
26-30	XO
31	Rc

- **M Form**

Bits	Value
0-5	OPCD
6-10	RS
11-15	RA
16-20	RB or SH
21-25	MB
26-30	ME
31	Rc

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.

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the `hello.s` assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Split-field notation:

The instruction field in some cases occupies more than one contiguous sequence of bits that are used in permuted order. Such a field is called a *split field*.

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.

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Instruction fields:

The set of instruction fields.

Item	Description				
AA (30)	Specifies an Absolute Address bit: <table><tr><td>0</td><td>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.</td></tr><tr><td>1</td><td>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).</td></tr></table>	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).
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.				

Item	Description																				
BO (6:10)	Specifies options for the branch conditional instructions. The possible encodings for the B0 field are: <table> <tr> <th>BO</th><th>Description</th></tr> <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>0100x</td><td>Decrement CTR. Branch if the decremented CTR value is not equal to 0 and the condition is true.</td></tr> <tr> <td>0101x</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> </table>	BO	Description	0000x	Decrement Count Register (CTR). Branch if the decremented CTR value is not equal to 0 and the condition is false.	0001x	Decrement CTR. Branch if the decremented CTR value is 0 and the condition is false.	001xx	Branch if the condition is false.	0100x	Decrement CTR. Branch if the decremented CTR value is not equal to 0 and the condition is true.	0101x	Decrement CTR. Branch if the decremented CTR value is equal to 0 and the condition is true.	011x	Branch if the condition is true.	1x00x	Decrement CTR. Branch if the decremented CTR value is not equal to 0.	1x01x	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).	1x1xx	Branch always.
BO	Description																				
0000x	Decrement Count Register (CTR). Branch if the decremented CTR value is not equal to 0 and the condition is false.																				
0001x	Decrement CTR. Branch if the decremented CTR value is 0 and the condition is false.																				
001xx	Branch if the condition is false.																				
0100x	Decrement CTR. Branch if the decremented CTR value is not equal to 0 and the condition is true.																				
0101x	Decrement CTR. Branch if the decremented CTR value is equal to 0 and the condition is true.																				
011x	Branch if the condition is true.																				
1x00x	Decrement CTR. Branch if the decremented CTR value is not equal to 0.																				
1x01x	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).																				
1x1xx	Branch always.																				
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> <tr> <th>Bit</th><th>Description</th></tr> <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> </table>	Bit	Description	7	FPSCR field 0 (bits 00:03)	8	FPSCR field 1 (bits 04:07)	9	FPSCR field 2 (bits 08:11)	10	FPSCR field 3 (bits 12:15)	11	FPSCR field 4 (bits 16:19)	12	FPSCR field 5 (bits 20:23)	13	FPSCR field 6 (bits 24:27)	14	FPSCR field 7 (bits 28:31)		
Bit	Description																				
7	FPSCR field 0 (bits 00:03)																				
8	FPSCR field 1 (bits 04:07)																				
9	FPSCR field 2 (bits 08:11)																				
10	FPSCR field 3 (bits 12:15)																				
11	FPSCR field 4 (bits 16:19)																				
12	FPSCR field 5 (bits 20:23)																				
13	FPSCR field 6 (bits 24:27)																				
14	FPSCR field 7 (bits 28:31)																				
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 mtrcf instruction: <table> <tr> <th>Bit</th><th>Description</th></tr> <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> </table>	Bit	Description	12	CR field 0 (bits 00:03)	13	CR field 1 (bits 04:07)	14	CR field 2 (bits 08:11)	15	CR field 3 (bits 12:15)	16	CR field 4 (bits 16:19)	17	CR field 5 (bits 20:23)	18	CR field 6 (bits 24:27)	19	CR field 7 (bits 28:31)		
Bit	Description																				
12	CR field 0 (bits 00:03)																				
13	CR field 1 (bits 04:07)																				
14	CR field 2 (bits 08:11)																				
15	CR field 3 (bits 12:15)																				
16	CR field 4 (bits 16:19)																				
17	CR field 5 (bits 20:23)																				
18	CR field 6 (bits 24:27)																				
19	CR field 7 (bits 28:31)																				
I (16:19)	Specifies the data to be placed into a field in the FPSCR. This is an immediate field.																				

Item	Description
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) = ones mask(all other) = zeros If mstart = mstop + 1 then mask(0:31) = ones If mstart > mstop + 1 then mask(mstop+1..mstart-1) = zeros mask(all other) = 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 S0 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 svc routine at address '1' LEV b'00000' 1 svc routine at address 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 mtspr and mfspr instructions. See the mtspr and mfspr instructions for information on the SPR encodings.
SR (11:15)	Specifies one of the 16 Segment Registers. Bit 11 is ignored.

Item	Description												
TO (6:10)	Specifies the conditions on which to trap. See Fixed-Point Trap Instructions for more information on condition encodings.												
	<table> <tr> <th>TO Bit</th><th>ANDed with Condition</th></tr> <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> </table>	TO Bit	ANDed with Condition	0	Compares less than.	1	Compares greater than.	2	Compares equal.	3	Compares logically less than.	4	Compares logically greater than.
TO Bit	ANDed with Condition												
0	Compares less than.												
1	Compares greater than.												
2	Compares equal.												
3	Compares logically less than.												
4	Compares logically greater than.												
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 concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Branch processor

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

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.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER® family and PowerPC® architecture overview” on page 10

A POWER® family or PowerPC® microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

Branch instructions

The branch instructions are used to change the sequence of instruction execution.

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 B0 field only or the B0 field and a partial BI field into the mnemonics.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER[®] family and PowerPC[®] architecture overview” on page 10

A POWER[®] family or PowerPC[®] microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

System call instruction

The PowerPC[®] system call instructions generate an interrupt or the system to perform a service.

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 (PowerPC[®])
- svc (Supervisor Call) instruction (POWER[®] family)

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER[®] family and PowerPC[®] architecture overview” on page 10

A POWER[®] family or PowerPC[®] microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Fixed-point processor”

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Extended mnemonics of condition register logical instructions” on page 126

The extended mnemonics of condition register logical instructions are available in POWER[®] family and PowerPC[®].

“sc (System Call) instruction” on page 405

“svc (Supervisor Call) instruction” on page 496

Condition register instructions

The condition register instructions copy one CR field to another CR field or perform logical operations on CR bits.

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.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER[®] family and PowerPC[®] architecture overview” on page 10

A POWER[®] family or PowerPC[®] microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Fixed-point processor”

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Extended mnemonics of condition register logical instructions” on page 126

The extended mnemonics of condition register logical instructions are available in POWER[®] family and PowerPC[®].

Fixed-point processor

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

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.

Related concepts:

“lhz (Load Half and Zero) instruction” on page 314

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER® family and PowerPC® architecture overview” on page 10

A POWER® family or PowerPC® microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

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 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.

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.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER® family and PowerPC® architecture overview” on page 10

A POWER® family or PowerPC® microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Extended mnemonics of condition register logical instructions” on page 126

The extended mnemonics of condition register logical instructions are available in POWER® family and PowerPC®.

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.

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.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER® family and PowerPC® architecture overview” on page 10

A POWER® family or PowerPC® microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

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.

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.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER® family and PowerPC® architecture overview” on page 10

A POWER® family or PowerPC® microprocessor contains a branch processor, a fixed-point processor, and

a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

Fixed-point address computation instructions

The different address computation instructions in POWER[®] family are merged into the arithmetic instructions for PowerPC[®].

There are several address computation instructions in POWER[®] family. These are merged into the arithmetic instructions for PowerPC[®].

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER[®] family and PowerPC[®] architecture overview” on page 10

A POWER[®] family or PowerPC[®] microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

Fixed-point arithmetic instructions

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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 condition register logical instructions” on page 126 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 for information on the available milicode routines.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER[®] family and PowerPC[®] architecture overview” on page 10

A POWER[®] family or PowerPC[®] microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

Fixed-point compare instructions

The fixed-point compare instructions algebraically or logically compare the contents of register RA.

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 arithmetic instructions” on page 127.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER[®] family and PowerPC[®] architecture overview” on page 10

A POWER[®] family or PowerPC[®] microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

Fixed-point trap instructions

Fixed-point trap instructions test for a specified set of conditions.

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:

TO Field Bit	ANDed with Condition
0	Less than
1	Greater than
2	Equal
3	Logically less than
4	Logically greater than

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.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER[®] family and PowerPC[®] architecture overview” on page 10

A POWER[®] family or PowerPC[®] microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Extended mnemonics in com mode” on page 148

The extended mnemonics for branch, logical, load, and arithmetic instructions.

Fixed-point logical instructions

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

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 condition register logical instructions” on page 126.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER[®] family and PowerPC[®] architecture overview” on page 10

A POWER[®] family or PowerPC[®] microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

Fixed-point rotate and shift instructions

The fixed-point rotate and shift instructions rotate the contents of a register.

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.

Related concepts:

“rlwinm or rlinm (Rotate Left Word Immediate Then AND with Mask) instruction” on page 399

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER® family and PowerPC® architecture overview” on page 10

A POWER® family or PowerPC® microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Extended mnemonics of 32-bit fixed-point rotate and shift instructions” on page 137

The extended mnemonics of 32-bit fixed-point rotate and shift instructions.

Fixed-point move to or from special-purpose registers instructions

The instructions move the contents of one Special-Purpose Register (SPR) into another SPR or into a General-Purpose Register (GPR). These include both nonprivileged and privileged 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 **mf spr** instruction or the **mt spr** instruction. The numbers defined in the encoding tables for the **mf spr** and **mt spr** instructions have already had their low-order 5 bits and high-order 5 bits reversed. When using the **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 **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 **mf dec** instruction has different values in PowerPC® and POWER® family. The DEC encoding number is 22 for PowerPC® and 6 for POWER® family. If the **mf dec** instruction is used, the assembler determines the DEC encoding based on the current assembly mode. The following list shows the assembler processing of the **mf dec** 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER[®] family and PowerPC[®] architecture overview” on page 10

A POWER[®] family or PowerPC[®] microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Floating-point processor”

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

Floating-point processor

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

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.

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER[®] family and PowerPC[®] architecture overview” on page 10

A POWER[®] family or PowerPC[®] microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

Floating-point numbers

A floating-point number consists of a signed significand and expresses a quantity that is the product of the signed fraction and the number.

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^{exponent} . Encodings are provided in the data format to represent:

- Finite numeric values
- +/- 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. The floating-point register is used to execute the instruction.

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.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER® family and PowerPC® architecture overview” on page 10

A POWER® family or PowerPC® microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

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

The Floating-point move instructions copy data from one FPR to another FPR.

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.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER® family and PowerPC® architecture overview” on page 10

A POWER® family or PowerPC® microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

Floating-point arithmetic instructions

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

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

Floating-point multiply-add instructions

The Floating-point multiply-add instructions combine a multiply operation and an add operation without an intermediate rounding operation.

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.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER® family and PowerPC® architecture overview” on page 10

A POWER® family or PowerPC® microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

Floating-point compare instructions

The Floating-point compare instructions perform ordered and unordered comparisons of the contents of two FPRs.

Floating-point compare instructions perform ordered and unordered comparisons of the contents of two FPRs. The CR field specified by the BF 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:

Item	Description	Description
Condition-Register Field and Floating-Point Condition Code Interpretation	Condition-Register Field and Floating-Point Condition Code Interpretation	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)

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER® family and PowerPC® architecture overview” on page 10

A POWER® family or PowerPC® microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

Floating-point conversion instructions

The Floating-point conversion instructions convert a floating-point operand in an FPR into a 32-bit signed fixed point register.

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.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER® family and PowerPC® architecture overview” on page 10

A POWER® family or PowerPC® microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

Floating-point status and control register instructions

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

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

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“POWER® family and PowerPC® architecture overview” on page 10

A POWER® family or PowerPC® microprocessor contains a branch processor, a fixed-point processor, and a floating-point processor.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

Syntax and semantics

The syntax and semantics of assembler language are defined.

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

Character set

There are defined characters in the operating system assembler language.

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:

Item	Description
, (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.

Item	Description
; (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: <pre>a 3,4,5 # These two lines have a 4,3,5 # the same effect as...</pre> <pre>a 3,4,5; a 4,3,5 # ...this line.</pre>
\$ (dollar sign)	Refers to the current value in the assembler's current location counter. For example: <pre>dino: .long 1,2,3 size: .long \$ - dino</pre>
@ (at sign)	Used after symbol names in expressions to specify explicit relocation types.

Related concepts:

“Reserved words”

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

“Expressions” on page 57

An expression is formed by one or more terms.

“Combining expressions with group 2 operators” on page 65

The assembler applies the rule of combining expressions with Group 2 operators.

Related information:

atof

Reserved words

There are no reserved words in the operating system assembler language.

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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“Constructing symbols” on page 40

The Symbols consist of numeric digits, underscores, periods, or lowercase letters.

“.extern pseudo-op” on page 535

“.globl pseudo-op” on page 538

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format”

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Line format

The assembler supports a free-line format for the source files.

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.

The assembly 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 concepts:

“Migration of branch conditional statements with no separator after mnemonic” on page 153

The AIX[®] assembler parses some statements different from the previous version of the assembler.

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Instruction statements and pseudo-operation statements

A instruction or pseudo-op statement has a predefined syntax.

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)

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Null statements

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

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 (**.csect**, **.comm**, and **.lcomm**, for example) may prevent a null statement's label from receiving the value of the address of the next statement.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Separator characters

The separator characters are spaces, tabs, and commas.

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.

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

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

“Migration of branch conditional statements with no separator after mnemonic” on page 153

The AIX[®] assembler parses some statements different from the previous version of the assembler.

Related information:

atof

Labels

The label entry is optional. The assembler gives the label the value contained in the assembler’s current location counter.

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.
```

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Mnemonics

The mnemonic field identifies whether a statement is an instruction statement or a pseudo-op statement.

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 ai (Add Immediate) or 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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

Related information:

atof

Operands

The existence and meaning of the operands depends on the mnemonic used.

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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Symbols”

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Comments

The comments option is optional in assembler.

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 concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols”

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

“.comm pseudo-op” on page 522

“.double pseudo-op” on page 527

“.toco pseudo-op” on page 560

Related information:

atof

Symbols

The symbol is used as a label operand.

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

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Constructing symbols

The Symbols consist of numeric digits, underscores, periods, or lowercase letters.

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. The maximum length of a symbol name is 65535 single-byte characters.

If a symbol must contain blanks or other special characters, the **.rename** pseudo-op allows a local name to be used as a synonym or alias for the global name.

With the exception of control section (csect) or Table of Contents (TOC) entry names, symbols might be used to represent storage locations or arbitrary data. The value of a symbol is always a 32-bit quantity when assembling the symbol in 32-bit mode, and the value is always a 64-bit quantity when assembling the symbol in 64-bit mode.

The following are valid examples of symbol names:

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

The following are not valid symbol names:

Item	Description
<code>7_sum</code>	(Begins with a digit.)
<code>#ofcredits</code>	(The # makes this a comment.)
<code>aa*1</code>	(Contains *, a special character.)
<code>IN AREA</code>	(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**, **.dssect**, **.csect**, or **.rename** pseudo-op.

Related concepts:

“.set pseudo-op” on page 551

“.comm pseudo-op” on page 522

“.dsect pseudo-op” on page 529

“.lcomm pseudo-op” on page 541

“.rename pseudo-op” on page 550

Defining a symbol with a label

The symbol can be defined by using it as a label.

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

```
                .using          dataval[RW],5
loop:           bgt             cont
.
.
                bdz             loop
cont:           l               3,dataval
                a               4,3,4
.
.
.csect 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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Defining a symbol with a pseudo-op

Use a symbol as the name operand of a pseudo-op to define the symbol.

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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

CSECT entry names

A symbol can be used as the *qualname* operand of the **csect** pseudo-op.

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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

“.csect pseudo-op” on page 525

Related information:

atof

Thread-local symbols

The two storage-mapping classes used for thread-local symbols are **TL** and **UL**.

The two storage-mapping classes that are used for thread-local symbols are **TL** and **UL**. The **TL** storage-mapping class is used with the **.csect** pseudo-op to define initialized thread-local storage. The **UL** storage-mapping class is used with the **.comm** or **.lcomm** pseudo-op to define thread-local storage that is not initialized. Expressions combining thread-local symbols and non-thread-local symbols are not allowed.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

The special symbol `toc`

The `.toc` pseudo-op creates the TOC anchor entry.

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 559.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

`atof`

TOC entry names

A symbol can be defined when used as the *Name* operand of the `.tc` pseudo-op.

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]
```

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

TOC-end symbols

TOC symbols use the storage-mapping class.

Most TOC symbols use the storage-mapping class TC. These symbols are collected at link time in an arbitrary order. If TOC overflow occurs, it is useful to move some TOC symbols to the end of the TOC and use an alternate code sequence to reference these symbols. Symbols to be moved to the end must use the storage-mapping class TE. Symbols with the TE storage-mapping class are treated identically to symbols with the TC storage-mapping class, except with respect to the RLDs chosen when the symbols are used in D-form instructions.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Using a symbol before defining it

A symbol can be referenced before using 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].
    l 5,ten(6)
    .
    .
.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 prints an error message unless the **-u** flag of the **as** command is used. External symbols might be declared in a statement by using the “.extern pseudo-op” on page 535.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Visibility of symbols

A visibility property can be associated with global symbols to be used by the linker when it creates a program or a shared object. The visibility of a symbol is specified with an optional parameter of the **.extern**, **.globl**, **.weak**, or **.comm** pseudo-op. The following visibility properties are defined for a symbol:

exported: The symbol is exported and preemptible.

protected: The symbol is exported but is not preemptible.

hidden: The symbol is not exported.

internal: The symbol cannot be exported.

Symbol visibilities are also affected by linker options.

Related concepts:

“.extern pseudo-op” on page 535

“.weak pseudo-op” on page 566

“.globl pseudo-op” on page 538

Related information:

ld

Declaring a global symbol

Global symbols, including external and weak symbols, participate in symbol resolution during the linking process. Other symbols are local and cannot be used outside the current source file. A symbol is declared global by using the **.extern**, **.globl**, **.weak**, or **.comm** pseudo-op.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants”

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Relocation specifiers

The relocation specifiers are used after symbol names or QualNames.

In general, the assembler generates the proper relocations based on expression types and usage. In some cases, however, multiple relocation types are possible, and an explicit relocation specifier is required.

An explicit relocation specifier can be used after symbol names or QualNames. It consists of the @ (at sign) character, and a 1- or 2-character relocation type. Relocation types can be specified in uppercase or lowercase letters, but not mixed case.

The following table lists the valid relocation types:

Type	RLD name	Usage
u	R_TOCU	Large TOC relocation
l	R_TOCL	Large TOC relocation
gd	R_TLS	Thread-Local Storage
ie	R_TLS_IE	Thread-Local Storage
le	R_TLS_LE	Thread-Local Storage
ld	R_TLS_LD	Thread-Local Storage
m	R_TLSM	Thread-Local Storage
ml	R_TLSML	Thread-Local Storage
tr	R_TRL	TOC references
tc	R_TOC	TOC references
p	R_POS	General

The large-TOC relocation types are used by default with the XMC_TE storage-mapping class, but they can also be used when TOC-relative instructions are used with XMC_TC symbols.

The thread-local storage relocation types are generally used with TOC references to thread-local symbols. The @gd relocation specifier is used by default. If other TLS access methods are used, an explicit specifier is needed.

The @tr relocation specifier enables the R_TRL relocation type to be used with TOC-relative loads. This relocation type prevents the linker from transforming the load instruction to an add-immediate instruction.

The @tc and @p relocation specifiers are never needed, but are provided for completeness.

Constants

The assembler language provides four kinds of constants.

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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Arithmetic constants

The assembler language provides arithmetic constants dependent upon the assembly mode.

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:

Constant Value	Valid/Invalid?
1_800_500	Valid
0xFFFFFFFF_00000000	Valid
0b111010_00100_00101_00000000001000_00	Valid (this is the "ld 4,8(5)" instruction)
0x_FFFF	Invalid

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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Arithmetic evaluation:

The arithmetic evaluation uses 32-bit mode and 64-bit mode.

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).

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Decimal constants:

Base 10 is the default base for arithmetic 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:

```
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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Octal constants:

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

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

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

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Hexadecimal constants:

To specify a hexadecimal number, prefix the number with 0X or 0x.

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.

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

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Binary constants:

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

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

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

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Floating-point constants:

A floating point constant has different components in the specified order.

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

Item	Description
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 **strtold** subroutine to perform the conversion to floating point. Check current documentation for restrictions and return values.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Character constants

Character constants are used when you want to use the code for a particular character.

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.
```

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Symbolic constants

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

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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

String constants

The string constants can be used as operands to certain pseudo-ops, such as **.rename**, **.byte**, and **.string** pseudo-ops.

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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address

(EA) into one of the GPRs.

Related information:

atof

Operators

The assembler provides three types of unary 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:

Item	Description
+	unary positive
-	unary negative
~	one's complement (unary)

The assembler provides the following binary operators:

Item	Description
*	multiplication
/	division
>	right shift
<	left shift
	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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators”

The assembler provides three types of unary operators.

Related information:

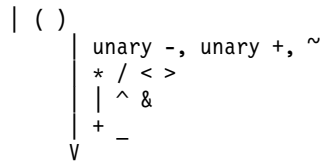
atof

Operator precedence

Operator precedence has 32-bit expressions.

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

Highest Priority



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:

Operation	Result	Remainder
8/3	2	2
8/-3	-2	2
(-8)/3	-2	-2
(-8)/(-3)	2	-2

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 concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Expressions

An expression is formed by one or more terms.

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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

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.

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 **.llong** pseudo-op, it is worthwhile to point out that expressions used to initialize the contents of a **.llong** may not require relocation. In 64-bit mode, relocation is applied to doubleword quantities. Thus, expression results that require relocation can not have their value stored in a location smaller than a doubleword.

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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Types and values of terms

The list of terms used and the value of the term.

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

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Absolute terms:

A term is absolute if its value does not change upon program relocation.

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 47).
- A symbol set to an absolute expression.

The value of an absolute term is the constant value.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Relocatable terms:

A term is relocatable if its value changes upon program relocation.

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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

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 file, or defined in the current file and global), 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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

TOC-relative relocatable terms:

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

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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

TOCOF relocatable terms:

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

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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Types and values of expressions

The types of terms and their value expression.

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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Restricted external relocatable expressions:

An expression has restricted external relocatable (E_REXT) type if it contains two relocatable terms.

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 64) and have opposite signs.

In a restricted external relocatable expression, a mixture of thread-local symbols and non-thread-local symbols is not allowed. That is, if one symbol has a thread-local storage-mapping class (TL or UL), the other symbol must have a thread-local storage-mapping class as well.

The following are examples of combinations of relocatable terms that produce an expression with restricted external relocatable type:

- <E_EXT> - <E_EXT>
- <E_REL> - <E_REL>
- <E_TREL> - <E_TREL>
- <E_EXT> - <E_REL>
- <E_REL> - <E_EXT>
- <E_EXT> - <E_TREL>
- <E_TREL> - <E_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.

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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Expression value calculations:

The rules are applied when calculating the values.

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 rules when determining the requirements for object file relocation table entries for an expression.

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 rules when determining the type of a combined expression.

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

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Combining expressions with group 1 operators:

The assembler applies the rule of combining expressions with Group 1 operators.

The following operators belong to group #1:

- *, /, >, <, |, &, ^

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.

Related concepts:

“Character set” on page 32

There are defined characters in the operating system assembler language.

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

Related information:

atof

Combining expressions with group 2 operators:

The assembler applies the rule of 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:

Item	Description
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:

Item	Description	
Type	Conditions for Expression to have Type	Relocation Table Entries
E_ABS	All the terms of the expression are paired relocatable terms, opposite terms, and absolute terms.	An RLD with type R_REF is generated for each opposite term.
E_REXT	The expression contains two unpaired relocatable terms with opposite signs in addition to all the paired relocatable terms, opposite terms, and absolute terms.	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. If one term has a thread-local storage-mapping class and the other does not, an error is reported.

Item	Description	
E_REL, E_EXT	The expression contains only one unpaired E_REL or E_RXT term in addition to all the paired relocatable terms, opposite terms, and absolute terms.	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.
E_TREL	The expression contains only one unpaired E_TREL term in addition to all the paired relocatable terms, opposite terms, and absolute terms.	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.
Error	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.	

The following example illustrates the preceding table:

```

.file "f1.s"
.csect A[PR]
L1: ai 10, 20, 30
L2: ai 10, 20, 30
EL1: l 10, 20(20)
     .extern EL1
     .extern EL2
EL2: l 10, 20(20)
     .csect B[PR]
BL1: l 10, 20(20)
BL2: l 10, 20(20)
     ba 16 + EL2 - L2 + L1          # Result is E_REL
     l 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 concepts:

“Reserved words” on page 33

There are no reserved words in the operating system assembler language.

“Line format” on page 34

The assembler supports a free-line format for the source files.

“Statements” on page 35

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.

“Symbols” on page 39

The symbol is used as a label operand.

“Constants” on page 47

The assembler language provides four kinds of constants.

“Operators” on page 55

The assembler provides three types of unary operators.

“.tc pseudo-op” on page 558

“.toc pseudo-op” on page 559

“.toctof pseudo-op” on page 560

Related information:

atof subroutine

Addressing

The assembly language uses different addressing modes and addressing considerations.

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

Absolute addressing

An absolute address is represented by the contents of a register.

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 concepts:

“Absolute addressing”

An absolute address is represented by the contents of a register.

“Explicit-based addressing” on page 70

Explicit-based addresses are specified as a base register number, *RA*, and a displacement, *D*

“Implicit-based addressing” on page 71

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.

“Location counter” on page 72

The assembler language program has a location counter used to assign storage addresses to your program’s statements.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“bcctr or bcc (Branch Conditional to Count Register) instruction” on page 179

“bclr or bcr (Branch Conditional Link Register) instruction” on page 182

“b (Branch) instruction” on page 176

“bc (Branch Conditional) instruction” on page 177

Absolute immediate addressing

An absolute immediate address is designated by immediate data.

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 concepts:

“Absolute addressing”

An absolute address is represented by the contents of a register.

“Location counter” on page 72

The assembler language program has a location counter used to assign storage addresses to your

program's statements.

"Branch processor" on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

"bclr or bcr (Branch Conditional Link Register) instruction" on page 182

"b (Branch) instruction" on page 176

"bc (Branch Conditional) instruction" on page 177

".using pseudo-op" on page 561

".drop pseudo-op" on page 528

Related information:

"Relative immediate addressing"

Relative immediate addresses are specified as immediate data within the object code and are calculated relative to the current instruction location.

Relative immediate addressing

Relative immediate addresses are specified as immediate data within the object code and are calculated relative to the current instruction location.

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 fullwords 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 concepts:

"Absolute addressing" on page 68

An absolute address is represented by the contents of a register.

"Absolute immediate addressing" on page 68

An absolute immediate address is designated by immediate data.

"Implicit-based addressing" on page 71

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.

"Location counter" on page 72

The assembler language program has a location counter used to assign storage addresses to your program's statements.

"Branch processor" on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

"bcctr or bcc (Branch Conditional to Count Register) instruction" on page 179

"bclr or bcr (Branch Conditional Link Register) instruction" on page 182

"b (Branch) instruction" on page 176

"bc (Branch Conditional) instruction" on page 177

".using pseudo-op" on page 561

".drop pseudo-op" on page 528

Related information:

"Relative immediate addressing"

Relative immediate addresses are specified as immediate data within the object code and are calculated relative to the current instruction location.

Explicit-based addressing

Explicit-based addresses are specified as a base register number, *RA*, and a displacement, *D*

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 **.comm** pseudo-op.

Note:

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 concepts:

“Absolute addressing” on page 68

An absolute address is represented by the contents of a register.

“Absolute immediate addressing” on page 68

An absolute immediate address is designated by immediate data.

“Implicit-based addressing” on page 71

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.

“Location counter” on page 72

The assembler language program has a location counter used to assign storage addresses to your program’s statements.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“bcctr or bcc (Branch Conditional to Count Register) instruction” on page 179

“bclr or bcr (Branch Conditional Link Register) instruction” on page 182

“.using pseudo-op” on page 561

“.drop pseudo-op” on page 528

Related information:

“Relative immediate addressing” on page 69

Relative immediate addresses are specified as immediate data within the object code and are calculated relative to the current instruction location.

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.

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 *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.

```
.toc
T.data: .tc data[tc],data[rw]
.csect data[rw]
foo: .long 2,3,4,5,6
```

```

bar: .long 777

.csect text[pr]
.align 2
l 10,T.data(2)    # Loads the address of
                  # csect data[rw] into GPR 10.

.using data[rw], 10    # Specify displacement.
l 3,foo             # The assembler generates l 3,0(10)
l 4,foo+4           # The assembler generates l 4,4(10)
l 5,bar             # The assembler generates l 5,20(10)

```

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

“.using pseudo-op” on page 561

Location counter

The assembler language program has a location counter used to assign storage addresses to your program’s statements.

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 concepts:

“Absolute addressing” on page 68

An absolute address is represented by the contents of a register.

“Absolute immediate addressing” on page 68

An absolute immediate address is designated by immediate data.

“Location counter”

The assembler language program has a location counter used to assign storage addresses to your program’s statements.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“bcctr or bcc (Branch Conditional to Count Register) instruction” on page 179

“bclr or bcr (Branch Conditional Link Register) instruction” on page 182

“Branch instructions” on page 20

The branch instructions are used to change the sequence of instruction execution.

“.using pseudo-op” on page 561

“.drop pseudo-op” on page 528

Assembling and linking a program

The assembly language program defines the commands for assembling and linking a program.

This section provides information on the following:

Related concepts:

“Understanding assembler passes” on page 75

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Related information:

as

Assembling and linking a program

Assembly language programs can be assembled with the **as** command or the **cc** command.

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

The **as** command invokes the assembler.

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

```
as [ -a Mode ] [ -oObjectFile ] [ -n Name ] [ -u ] [ -lListFile ] [ -W | -w ] [ -xXCrossFile ] [ -s ListFile ] [ -m ModeName ] [ -M ] [ -Eoff|on ] [ -poff|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:

Item	Description
OBJECT_MODE=32	Produce 32-bit object code. The default machine setting is com .
OBJECT_MODE=64	Produce 64-bit object code (XCOFF64 files). The default machine setting is ppc64 .
OBJECT_MODE=32_64	Invalid.
OBJECT_MODE= <i>anything else</i>	Invalid.

Related information:

as

Assembling and linking with the cc command

The **cc** command can be used to assemble and link an assembly source program.

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 (**ld** command) invoked by the **cc** command searches for **file.o**.

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

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

For more information on the option flags passed to the **cc** command, see the **cc** command.

Related concepts:

“Understanding assembler passes” on page 75

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Related information:

as

Understanding assembler passes

When you enter the **as** command, the assembler makes two passes over the source program.

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

First pass

During the first pass, the assembler checks to see if the instructions are legal in the current assembly mode.

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 **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.

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Second pass

During 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.

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 concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing”

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Related information:

as

Interpreting an assembler listing

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

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:

```
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:

```
hello.s          V4.0          01/25/1994
File# Line#  Mode Name      Loc Ctr  Object Code      Source
0         1  |
0         2  |          #####
0         3  |          # C source code
0         4  |          #####
0         5  |          #   hello()
0         6  |          #   {
0         7  |          #   printf("hello,world\n");
0         8  |          #   }
0         9  |          #####
0        10  |          #   Compile as follows:
0        11  |          #   cc -o helloworld hello.s
0        12  |          #
0        13  |          #####
0        14  |          .file "hello.s"
0        15  |          #Static data entry in
0        16  |          #T(able)O(f)C(ontents)
0        17  |          .toc
0        18  |          T.data:  .tc data[tc],data[rw]
0        19  |          .globl main[ds]
0         20  |          #main[ds] contains definitions for
```

```

0 20 |                                     #runtime linkage of function main
0 21 |                                     .csect    main[ds]
0 22 | COM main 00000000 00000000         .long    .main[PR]
0 23 | COM main 00000004 00000050         .long    TOC[tc0]
0 24 | COM main 00000008 00000000         .long    0
0 25 |                                     #Function entry in
0 26 |                                     #T(able)O(f)C(ontents)
0 27 |                                     .toc
0 28 | COM .main 00000000 00000034         T.hello: .tc .main[tc],main[ds]
0 29 |                                     .globl    .main[PR]
0 30 |
0 31 |                                     #Set routine stack variables
0 32 |                                     #Values are specific to
0 33 |                                     #the current routine and can
0 34 |                                     #vary from routine to routine
0 35 |                                     00000020 .set    argarea,    32
0 36 |                                     00000018 .set    linkarea,   24
0 37 |                                     00000000 .set    locstckarea, 0
0 38 |                                     00000001 .set    ngprs,      1
0 39 |                                     00000000 .set    nfprs,      0
0 40 |                                     0000003c .set    szdsa,      8*nfprs+4*ngprs+linkarea+
                                           argarea+locstckarea

0 41 |
0 42 |                                     #Main routine
0 43 |                                     .csect          .main[PR]
0 44 |
0 45 |
0 46 |                                     #PROLOG: Called Routines
0 47 |                                     #      Responsibilities
0 48 |                                     #Get link reg.
0 49 | COM .main 00000000 7c0802a6         mflr    0
0 50 |                                     #Not required to Get/Save CR
0 51 |                                     #because current routine does
0 52 |                                     #not alter it.
0 53 |
0 54 |                                     #Not required to Save FPR's
0 55 |                                     #14-31 because current routine
0 56 |                                     #does not alter them.
0 57 |
0 58 |                                     #Save GPR 31.
0 59 | COM .main 00000004 bfe1fffc         stm     31, -8*nfprs-4*ngprs(1)
0 60 |                                     #Save LR if non-leaf routine.
0 61 | COM .main 00000008 90010008         st      0, 8(1)
0 62 |                                     #Decrement stack ptr and save
0 63 |                                     #back chain.
0 64 | COM .main 0000000c 9421ffc4         stu     1, -szdsa(1)
0 65 |
0 66 |
0 67 |                                     #Program body
0 68 |                                     #Load static data address
0 69 | COM .main 00000010 81c20000         l       14,T.data(2)
0 70 |                                     #Line 3, file hello.c
0 71 |                                     #Load address of data string
0 72 |                                     #from data addr.
0 73 |                                     #This is a parameter to printf()
0 74 | COM .main 00000014 386e0000         cal     3,_helloworld(14)
0 75 |                                     #Call printf function
0 76 | COM .main 00000018 4bffffe9         bl      .printf[PR]
0 77 | COM .main 0000001c 4def7b82         cror    15, 15, 15
0 78 |
0 79 |
0 80 |                                     #EPILOG: Return Sequence
0 81 |                                     #Get saved LR.
0 82 | COM .main 00000020 80010044         l       0, szdsa+8(1)
0 83 |
0 84 |                                     #Routine did not save CR.
0 85 |                                     #Restore of CR not necessary.

```

```

0      86 |
0      87 |                                     #Restore stack ptr
0      88 | COM .main 00000024 3021003c         ai    1, 1, szdsa
0      89 |                                     #Restore GPR 31.
0      90 | COM .main 00000028 bbe1fffc         lm    31, -8*nfprs-4*ngprs(1)
0      91 |
0      92 |                                     #Routine did not save FPR's.
0      93 |                                     #Restore of FPR's not necessary.
0      94 |
0      95 |                                     #Move return address
0      96 |                                     #to Link Register.
0      97 | COM .main 0000002c 7c0803a6         mtlr0
0      98 |                                     #Return to address
0      99 |                                     #held in Link Register.
0     100 | COM .main 00000030 4e800021         brl
0     101 |
0     102 |
0     103 |                                     #External variables
0     104 |                                     .extern.printf[PR]
0     105 |
0     106 |                                     #####
0     107 |                                     #Data
0     108 |                                     #####
0     109 |                                     #String data placed in
0     110 |                                     #static csect data[rw]
0     111 |                                     .csect data[rw]
0     112 |                                     .align2
0     113 |                                     _helloworld:
0     114 | COM data 00000000 68656c6c         .byte 0x68,0x65,0x6c,0x6c
0     115 | COM data 00000004 6f2c776f         .byte 0x6f,0x2c,0x77,0x6f
0     116 | COM data 00000008 726c640a         .byte 0x72,0x6c,0x64,0xa,0x0
0     116 | COM data 0000000c 00

```

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:

Item	Description
File#	Lists the source file number. Files included with the M4 macro processor (-I option) are displayed by the number of the file in which the statement was found.
Line#	Refers to the line number of the assembler source code.
Mode	Indicates the current assembly mode for this instruction.
Name	Lists the name of the csect where this line of source code originates.
Loc Ctr	Lists the value contained in the assembler's location counter. The listing shows a location counter value only for assembler language instructions that generate object code.
Object Code	Shows the hexadecimal representation of the object code generated by each line of the assembler program. Since each instruction is 32 bits, each line in the assembler listing shows a maximum of 4 bytes. Any remaining bytes in a line of assembler source code are shown on the following line or lines. Note: If pass two failed, the assembler listing will not contain object code.
Source	Lists the assembler source code for the program. A limit of 100 ASCII characters will be displayed per line.

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.

```

L_dfmt_1.s                                V4.0                                01/26/1994
File# Line# Mode Name Loc Ctr Object Code PowerPC Source
0      1
0      2
0      3
0      4
0      5
0      6 COM dfmt 00000000 8025000c lwz l1,d1 # 8025000c
0      7 COM dfmt 00000004 b8c50018 lmw l1,6,d0 # b8c50018
0      8 COM dfmt 00000008 b0e50040 sth l1,7,d8 # b0e50040
0      9 COM dfmt 0000000c 80230020 lwz l1,1,0x20(3) # 80230020
0     10 COM dfmt 00000010 30220003 addic ai,1,2,3 # 30220003
0     11 COM dfmt 00000014 0cd78300 twi ti,6,23,-32000 # 0cd78300
0     12 COM dfmt 00000018 2c070af0 cmpi 0,7,2800 # 2c070af0
0     13 COM dfmt 0000001c 2c070af0 cmpi 0,0,7,2800 # 2c070af0
0     14 COM dfmt 00000020 30220003 subic si,1,2,-3 # 30220003
0     15 COM dfmt 00000024 34220003 subic. si,1,2,-3 # 34220003
0     16 COM dfmt 00000028 703e00ff andi. andil.30,1,0xFF # 703e00ff
0     17 COM dfmt 0000002c 2b9401f4 cmpli 7,20,500 # 2b9401f4
0     18 COM dfmt 00000030 0c2501a4 twlgti tlgti 5,420 # 0c2501a4
0     19 COM dfmt 00000034 34220003 addic. ai,1,2,3 # 34220003
0     20 COM dfmt 00000038 2c9ff380 cmpi 1,31,-3200 # 2c9ff380
0     21 COM dfmt 0000003c 281f0c80 cmpli 0,31,3200 # 281f0c80
0     22 COM dfmt 00000040 8ba5000c lbz 29,d1 # 8ba5000c
0     23 COM dfmt 00000044 85e5000c lwzu lu,15,d1 # 85e5000c
0     24 COM dfmt 00000048 1df5fec0 mulli muli 15,21,-320 # 1df5fec0
0     25 COM dfmt 0000004c 62af0140 ori oril 15,21,320 # 62af0140
0     26 COM dfmt 00000050 91e5000c stw st,15,d1 # 91e5000c
0     27 COM dfmt 00000054 bde5000c stmw stm,15,d1 # bde5000c
0     28 COM dfmt 00000058 95e5000c stwu stu,15,d1 # 95e5000c
0     29 COM dfmt 0000005c 69ef0960 xori xoril 15,15,2400 # 69ef0960
0     30 COM dfmt 00000060 6d8c0960 xoris xoriu 12,12,2400 # 6d8c0960
0     31 COM dfmt 00000064 3a9eff38 addi 20,30,-200 # 3a9eff38
0     32
0     33
0     34
0     35 COM also 00000000 00000000 .csect also[RW]
0                                     data:
0                                     .long 0,0,0
0     36 COM also 00000008 00000000
0     COM also 0000000c 00000003 d1:.long 3,4,5 # d1 = 0xC = 12
0     COM also 00000010 00000004
0     COM also 00000014 00000005
0     37 COM also 00000018 00000068 d0: .long data # d0 = 0x18 = 24
0     38 COM also 0000001c 00000000 data2: .space 36
0                                     00000020 ....
0     39 COM also 0000003c 00000000 d8: .long 9184 # d8 = 0x40 = 64
0     40 COM also 00000044 ffffffff d9: .long 0xFFFFFFFF # d9 = 0x44
0     41 #
0     42 # 0000 00000000 00000000 00000000 00000000
0     43 # 0010 00000004 00000005 0000000C 00000000
0     44 # 0020 00000000 00000000 00000000 00000000
0     45 # 0030 000023E0

```

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.

```

L_dfmt_2.s                                V4.0                                01/26/1994
File# Line# Mode Name Loc Ctr Object Code POWER Source
0      1 |                                     #%% -L

```

```

0      2      |                                     .machine "pwr"
0      3      |                                     .csect dfmt[PR]
0      4      |                                     .using data,5
0      5      | PWR dfmt 00000000 8025000c      l      lwz      1,d1
0      6      | PWR dfmt 00000004 b8650018      lm     lmw      3,d0
0      7      | PWR dfmt 00000008 b0e50040      sth     7,d8
0      8      | PWR dfmt 0000000c 80230020      l      lwz      1,0x20(3)
0      9      | PWR dfmt 00000010 30220003      ai     addic   1,2,3
0     10      | PWR dfmt 00000014 0cd78300      ti     twi     6,23,-32000
0     11      | PWR dfmt 00000018 2c070af0      cmpi   0,7,2800
0     12      | PWR dfmt 0000001c 2c070af0      cmpi   0,0,7,2800
0     13      | PWR dfmt 00000020 30220003      si     subic   1,2,-3
0     14      | PWR dfmt 00000024 34220003      si.    subic.  1,2,-3
0     15      | PWR dfmt 00000028 703e00ff      andil. andi.   30,1,0xFF
0     16      | PWR dfmt 0000002c 2b9401f4      cmpli  7,20,500
0     17      | PWR dfmt 00000030 0c2501a4      tlgti  twlgti  5,420
0     18      | PWR dfmt 00000034 34220003      ai.    addic.  1,2,3
0     19      | PWR dfmt 00000038 2c9ff380      cmpi   1,31,-3200
0     20      | PWR dfmt 0000003c 281f0c80      cmpli  0,31,3200
0     21      | PWR dfmt 00000040 8ba5000c      lbz    29,d1
0     22      | PWR dfmt 00000044 85e5000c      lu     lwzu    15,d1
0     23      | PWR dfmt 00000048 1df5fec0      muli   mulli   15,21,-320
0     24      | PWR dfmt 0000004c 62af0140      oril   ori     15,21,320
0     25      | PWR dfmt 00000050 91e5000c      st     stw     15,d1
0     26      | PWR dfmt 00000054 bde5000c      stm    stmw    15,d1
0     27      | PWR dfmt 00000058 95e5000c      stu    stwu    15,d1
0     28      | PWR dfmt 0000005c 69ef0960      xoril  xori    15,15,2400
0     29      | PWR dfmt 00000060 6d8c0960      xoriu  xoris   12,12,2400
0     30      | PWR dfmt 00000064 3a9eff38      addi   20,30,-200
0     31
0     32
0     33      |                                     .csect also[RW]
0     34
0     35      | PWR also 00000000 00000000      data:  .long   0,0,0
0     36      | PWR also 00000004 ....
0     36      | PWR also 00000008 00000000      d1: long 3,4,5
0     36      | PWR also 0000000c 00000003      # d1 = 0xc = 12
0     36      | PWR also 00000010 00000004
0     36      | PWR also 00000014 00000005
0     37      | PWR also 00000018 00000068      d0: long data # d0 = 0x18 = 24
0     38      | PWR also 0000001c 00000000      data2: space 36
0     38      | PWR also 00000020 ....
0     38      | PWR also 0000003c 00000000
0     39      | PWR also 00000040 000023e0      d8: long 9184 # d8 = 0x40 = 64
0     40      | PWR also 00000044 ffffffff      d9: long 0xFFFFFFFF # d9 = 0x44
0     41      | #
0     42      | # 0000 00000000 00000000 00000000 00000003
0     43      | # 0010 00000004 00000005 0000000c 00000000
0     44      | # 0020 00000000 00000000 00000000 00000000
0     45      | # 0030 000023E0

```

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Related information:

as

Interpreting a symbol cross-reference

The example of the symbol cross-reference for the **hello.s** assembly program.

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

Symbol	File	CSECT	Line #
.main	hello.s	--	22
.main	hello.s	.main	28 *
.main	hello.s	--	29
.main	hello.s	.main	43 *
.printf	hello.s	--	76
.printf	hello.s	--	104
T.data	hello.s	data	17 *
T.data	hello.s	data	69
T.hello	hello.s	.main	28 *
TOC	hello.s	TOC	23
_helloworld	hello.s	--	74
_helloworld	hello.s	data	113 *
argarea	hello.s	--	35 *
argarea	hello.s	--	40
data	hello.s	--	17
data	hello.s	data	17 *
data	hello.s	data	111 *
linkarea	hello.s	--	36 *
linkarea	hello.s	--	40
locstckarea	hello.s	--	37 *
locstckarea	hello.s	--	40
main	hello.s	--	18
main	hello.s	main	21 *
main	hello.s	main	28
nfprs	hello.s	--	39 *
nfprs	hello.s	--	40
nfprs	hello.s	--	59
nfprs	hello.s	--	90
ngprs	hello.s	--	38 *
ngprs	hello.s	--	40
ngprs	hello.s	--	59
ngprs	hello.s	--	90
szdsa	hello.s	--	40 *
szdsa	hello.s	--	64
szdsa	hello.s	--	82
szdsa	hello.s	--	88

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 concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the `hello.s` assembly program.

“Subroutine linkage convention”

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Related information:

`as`

Subroutine linkage convention

The Subroutine Linkage Convention.

This article discusses the following:

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the `hello.s` assembly program.

“Subroutine linkage convention”

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Linkage convention overview

The subroutine linkage convention describes the machine state at subroutine entry and exit.

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.

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the `hello.s` assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Object mode considerations:

The object mode considerations applies to both 32-bit mode and 64-bit mode.

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 (doubleword). 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 doubleword) 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 doubleword), 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 `ld` and the store instruction would be `std`.

Register usage and conventions:

The PowerPC® 32-bit architecture has 32 GPRs and 32PRs.

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 2. General-Purpose Register Conventions

Register	Status	Use
GPR0	volatile	In function prologs.
GPR1	dedicated	Stack pointer.
GPR2	dedicated	Table of Contents (TOC) pointer.
GPR3	volatile	First word of a function's argument list; first word of a scalar function return.
GPR4	volatile	Second word of a function's argument list; second word of a scalar function return.
GPR5	volatile	Third word of a function's argument list.
GPR6	volatile	Fourth word of a function's argument list.

Table 2. General-Purpose Register Conventions (continued)

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

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

Register	Status	Use
FPR0	volatile	As a scratch register.
FPR1	volatile	First floating-point parameter; first 8 bytes of a floating-point scalar return.
FPR2	volatile	Second floating-point parameter; second 8 bytes of a floating-point scalar return.
FPR3	volatile	Third floating-point parameter; third 8 bytes of a floating-point scalar return.
FPR4	volatile	Fourth floating-point parameter; fourth 8 bytes of a floating-point scalar return.
FPR5	volatile	Fifth floating-point parameter.
FPR6	volatile	Sixth floating-point parameter.
FPR7	volatile	Seventh floating-point parameter.
FPR8	volatile	Eighth floating-point parameter.
FPR9	volatile	Ninth floating-point parameter.
FPR10	volatile	Tenth floating-point parameter.
FPR11	volatile	Eleventh floating-point parameter.
FPR12	volatile	Twelfth floating-point parameter.
FPR13	volatile	Thirteenth floating-point parameter.
FPR14:FPR31	nonvolatile	If modified, must be preserved across a call.

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.

Example of calling convention for complex types

```
complex double foo(complex double);
```

Arguments are passed into fp1 and fp2 and the results are returned in fp1 and fp2. Subsequent complex double parameters are passed in the next two available registers, up to fp13, by using either even-odd or odd-even pairs. After fp13 they are passed in a parameter area in the memory located in the beginning of the caller's stack frame.

Note: The skipped registers are not used for later parameters. In addition, these registers are not initialized by the caller and the called function must not depend on the value stored within the skipped registers.

A single precision complex (complex float) is passed the same way as double precision with the values widened to double precision.

Double and single precision complex (complex double and complex float) are returned in fp1 and fp2 with single precision values widened to double precision.

Quadruple precision complex (complex long double) parameters are passed in the next four available registers, from fp1 to fp13 and then in the parameter area. The order in which the registers fill is, upper half of the real part, lower half of the real part, upper half of the imaginary part, and lower half of the imaginary part.

Note: In AIX structs, classes and unions are passed in gprs (or memory) and not fprs. This is true even if the classes and unions contain floating point values. In Linux on PPC the address of a copy in memory is passed in the next available gpr (or in memory). The varargs parameters are specifically handled and generally passed to both fprs and gprs.

Calling convention for decimal floating-point types (_Decimal128)

_Decimal64 parameters are passed in the next available fpr and the results returned in fp1.

_Decimal32 parameters are passed in the lower half of the next available fpr and the results are returned in the lower half of fp1, without being converted to _Decimal64.

_Decimal128 parameters are passed in the next available even-odd fpr pair (or memory) even if that means skipping a register and the results are returned in the even-odd pair fpr2 and fpr3. The reason is that all the arithmetic instructions require the use of even-odd register pairs.

Note: The skipped registers are not used for later parameters. In addition, these registers are not initialized by the caller and the called function must not depend on the value stored within the skipped registers.

Unlike float or double, with DFP, a function prototype is always required. Hence _Decimal32 is not required to be widened to _Decimal64.

Example of calling convention for decimal floating-point type (_Decimal32)

```
#include <float.h>
#define DFP_ROUND_HALF_UP 4

_Decimal32 Add_GST_and_Ontario_PST_d32 (_Decimal32 price)
{
```

```

Decimal32 gst;
Decimal32 pst;
Decimal32 total;
long original_rounding_mode = __dfp_get_rounding_mode ( );
__dfp_set_rounding_mode (DFP_ROUND_HALF_UP);
gst = price * 0.06dd;
pst = price * 0.08dd;
total = price + gst + pst;
__dfp_set_rounding_mode (original_rounding_mode);
return (total);
}

| 000000 PDEF Add_GST_and_Ontario_PST_d32
>> 0| PROC price,fp1
0 000000 stw 93E1FFFC 1 ST4A #stack(gr1,-4)=gr31
0 000004 stw 93C1FFF8 1 ST4A #stack(gr1,-8)=gr30
0 000008 stwu 9421FF80 1 ST4U gr1,#stack(gr1,-128)=gr1
0 00000C lwz 83C20004 1 L4A gr30=+CONSTANT_AREA(gr2,0)
0 000010 addi 38A00050 1 LI gr5=80
0 000014 ori 60A30000 1 LR gr3=gr5
>> 0| 000018 stfiwx 7C211FAE 1 STDFS price(gr1,gr3,0)=fp1
9 00001C mffs FC00048E 1 LFFSCR fp0=fcr
9 000020 stfd D8010058 1 STFL #MX_SET1(gr1,88)=fp0
9 000024 lwz 80010058 1 L4A gr0=#MX_SET1(gr1,88)
9 000028 rlwinm 5400077E 1 RN4 gr0=gr0,0,0x7
9 00002C stw 9001004C 1 ST4A original_rounding_mode(gr1,76)=gr0
10 000030 mtfsfi FF81410C 1 SETDRND fcr=4,fcr
11 000034 ori 60A30000 1 LR gr3=gr5
11 000038 lfiwax 7C011EAE 1 LDFS fp0=price(gr1,gr3,0)
11 00003C dctdp EC000204 1 CVDSL fp0=fp0,fcr
11 000040 lfd C83E0000 1 LDFL fp1=+CONSTANT_AREA(gr30,0)
11 000044 dmul EC000844 1 MDL fp0=fp0,fp1,fcr
11 000048 drsp EC000604 1 CVDLDS fp0=fp0,fcr
11 00004C addi 38600040 1 LI gr3=64
11 000050 ori 60640000 1 LR gr4=gr3
11 000054 stfiwx 7C0127AE 1 STDFS gst(gr1,gr4,0)=fp0
12 000058 ori 60A40000 1 LR gr4=gr5
12 00005C lfiwax 7C0126AE 1 LDFS fp0=price(gr1,gr4,0)
12 000060 dctdp EC000204 1 CVDSL fp0=fp0,fcr
12 000064 lfd C83E0008 1 LDFL fp1=+CONSTANT_AREA(gr30,8)
12 000068 dmul EC000844 1 MDL fp0=fp0,fp1,fcr
12 00006C drsp EC000604 1 CVDLDS fp0=fp0,fcr
12 000070 addi 38800044 1 LI gr4=68
12 000074 ori 60860000 1 LR gr6=gr4
12 000078 stfiwx 7C0137AE 1 STDFS pst(gr1,gr6,0)=fp0
13 00007C lfiwax 7C012EAE 1 LDFS fp0=price(gr1,gr5,0)
13 000080 lfiwax 7C211EAE 1 LDFS fp1=gst(gr1,gr3,0)
13 000084 dctdp EC000204 1 CVDSL fp0=fp0,fcr
13 000088 dctdp EC200A04 1 CVDSL fp1=fp1,fcr
13 00008C mffs FC40048E 1 LFFSCR fp2=fcr
13 000090 stfd D8410058 1 STFL #MX_SET1(gr1,88)=fp2
13 000094 lwz 80010058 1 L4A gr0=#MX_SET1(gr1,88)
13 000098 rlwinm 5400077E 1 RN4 gr0=gr0,0,0x7
13 00009C mtfsfi FF81710C 1 SETDRND fcr=7,fcr
13 0000A0 dadd EC000804 1 ADL fp0=fp0,fp1,fcr
13 0000A4 stw 90010058 1 ST4A #MX_CONV1_0(gr1,104)=gr3
13 0000BC stw 9001006C 1 ST4A #MX_CONV1_0(gr1,108)=gr0
13 0000C0 lfd C8210068 1 LDFL fp1=#MX_CONV1_0(gr1,104)
13 0000AC mtfsf FC030D8E 1 LFSCR8 fsr,fcr=fp1,1,1
13 0000B0 addi 38000007 1 LI gr0=7
13 0000B4 addi 38600000 1 LI gr3=0
13 0000B8 stw 90610068 1 ST4A #MX_CONV1_0(gr1,104)=gr3
13 0000BC stw 9001006C 1 ST4A #MX_CONV1_0(gr1,108)=gr0
13 0000C0 lfd C8210068 1 LDFL fp1=#MX_CONV1_0(gr1,104)
13 0000C4 drrnd EC010646 1 RRDFL fp0=fp0,fp1,3,fcr
13 0000C8 drsp EC000604 1 CVDLDS fp0=fp0,fcr
13 0000CC lfiwax 7C2126AE 1 LDFS fp1=pst(gr1,gr4,0)
13 0000D0 dctdp EC000204 1 CVDSL fp0=fp0,fcr

```

```

13 0000D4 dctdp EC200A04 1 CVDSL fp1=fp1, fcr
13 0000D8 mffs FC40048E 1 LFFSCR fp2=fcr
13 0000DC stfd D8410058 1 STFL #MX_SET1(gr1,88)=fp2
13 0000E0 lwz 80810058 1 L4A gr4=#MX_SET1(gr1,88)
13 0000E4 rlwinm 5484077E 1 RN4 gr4=gr4,0,0x7
13 0000E8 mtfsfi FF81710C 1 SETDRND fcr=7, fcr
13 0000EC dadd EC000804 1 ADFL fp0=fp0, fp1, fcr
13 0000F0 stw 90810058 1 ST4A #MX_SET1(gr1,88)=gr4
13 0000F4 lfd C8210058 1 LFL fp1=#MX_SET1(gr1,88)
13 0000F8 mtfsf FC030D8E 1 LFSCR8 fcr=fp1,1,1
13 0000FC stw 90610068 1 ST4A #MX_CONVF1_0(gr1,104)=gr3
13 000100 stw 9001006C 1 ST4A #MX_CONVF1_0(gr1,108)=gr0
13 000104 lfd C8210068 1 LDFL fp1=#MX_CONVF1_0(gr1,104)
13 000108 drrnd EC010646 1 RRDFL fp0=fp0, fp1,3, fcr
13 00010C drsp EC000604 1 CVDLDS fp0=fp0, fcr
13 000110 addi 38600048 1 LI gr3=72
13 000114 ori 60640000 1 LR gr4=gr3
13 000118 stfiwx 7C0127AE 1 STDFS total(gr1,gr4,0)=fp0
14 00011C lwz 8001004C 1 L4A gr0=original_rounding_mode(gr1,76)
14 000120 stw 90010058 1 ST4A #MX_SET1(gr1,88)=gr0
14 000124 lfd C8010058 1 LFL fp0=#MX_SET1(gr1,88)
14 000128 mtfsf FC03058E 1 LFSCR8 fcr=fp0,1,1
>> 15 00012C lfiwx 7C211EAE 1 LDfs fp1=total(gr1,gr3,0)
16 CL.1:
16 000130 lwz 83C10078 1 L4A gr30=#stack(gr1,120)
16 000134 addi 38210080 1 AI gr1=gr1,128
16 000138 bclr 4E800020 1 BA lr

```

Example of calling convention for decimal floating-point type (_Decimal64)

```

#include <float.h>
#define DFP_ROUND_HALF_UP 4

_Decimal64 Add_GST_and_Ontario_PST_d64 (_Decimal64 price)
{
    _Decimal64 gst;
    _Decimal64 pst;
    _Decimal64 total;
    Tong original_rounding_mode = __dfp_get_rounding_mode ( );
    __dfp_set_rounding_mode (DFP_ROUND_HALF_UP);
    gst = price * 0.06dd;
    pst = price * 0.08dd;
    total = price + gst + pst;
    __dfp_set_rounding_mode (original_rounding_mode);
    return (total);
}

| 000000 PDEF Add_GST_and_Ontario_PST_d64
>> 0| PROC price,fp1
0 000000 stw 93E1FFFC 1 ST4A #stack(gr1,-4)=gr31
0 000004 stw 93C1FFF8 1 ST4A #stack(gr1,-8)=gr30
0 000008 stwu 9421FF80 1 ST4U gr1,#stack(gr1,-128)=gr1
0 00000C lwz 83C20004 1 L4A gr30=+CONSTANT_AREA(gr2,0)
>> 0| 000010 stfd D8210098 1 STDFL price(gr1,152)=fp1
9 000014 mffs FC00048E 1 LFFSCR fp0=fcr
9 000018 stfd D8010060 1 STFL #MX_SET1(gr1,96)=fp0
9 00001C lwz 80010060 1 L4A gr0=#MX_SET1(gr1,96)
9 000020 rlwinm 5400077E 1 RN4 gr0=gr0,0,0x7
9 000024 stw 90010058 1 ST4A original_rounding_mode(gr1,88)=gr0
10 000028 mtfsfi FF81410C 1 SETDRND fcr=4, fcr
11 00002C lfd C8010098 1 LDFL fp0=price(gr1,152)
11 000030 lfd C83E0000 1 LDFL fp1=+CONSTANT_AREA(gr30,0)
11 000034 dmul EC000844 1 MDFL fp0=fp0, fp1, fcr
11 000038 stfd D8010040 1 STDFL gst(gr1,64)=fp0
12 00003C lfd C8010098 1 LDFL fp0=price(gr1,152)
12 000040 lfd C83E0008 1 LDFL fp1=+CONSTANT_AREA(gr30,8)
12 000044 dmul EC000844 1 MDFL fp0=fp0, fp1, fcr

```

```

12| 000048 stfd D8010048 1 STDPL pst(gr1,72)=fp0
13| 00004C lfd C8010098 1 LDPL fp0=price(gr1,152)
13| 000050 lfd C8210040 1 LDPL fp1=gst(gr1,64)
13| 000054 dadd EC000804 1 ADPL fp0=fp0,fp1,fcf
13| 000058 lfd C8210048 1 LDPL fp1=pst(gr1,72)
13| 00005C dadd EC000804 1 ADPL fp0=fp0,fp1,fcf
13| 000060 stfd D8010050 1 STDPL total(gr1,80)=fp0
14| 000064 lwz 80010058 1 L4A gr0=original_rounding_mode(gr1,88)
14| 000068 stw 90010060 1 ST4A #MX_SET1(gr1,96)=gr0
14| 00006C lfd C8010060 1 LPL fp0=#MX_SET1(gr1,96)
14| 000070 mtfsc FC03058E 1 LFSCR8 fsc,fcf=fp0,1,1
>> 15| 000074 lfd C8210050 1 LDPL fp1=total(gr1,80)
16| CL.1:
16| 000078 lwz 83C10078 1 L4A gr30=#stack(gr1,120)
16| 00007C addi 38210080 1 AI gr1=gr1,128
16| 000080 bclr 4E800020 1 BA lr

```

Example of calling convention for decimal floating-point type (_Decimal128)

```

include <float.h>
#define DFP_ROUND_HALF_UP 4

_Decimal128 Add_GST_and_Ontario_PST_d128 (_Decimal128 price)
{
    _Decimal128 gst;
    _Decimal128 pst;
    _Decimal128 total;
    long original_rounding_mode = __dfp_get_rounding_mode ( );
    __dfp_set_rounding_mode (DFP_ROUND_HALF_UP);
    gst = price * 0.06dd;
    pst = price * 0.08dd;
    total = price + gst + pst;
    __dfp_set_rounding_mode (original_rounding_mode);
    return (total);
}

| 000000 PDEF Add_GST_and_Ontario_PST_d128
>> 0| PROC price,fp2,fp3
0| 000000 stw 93E1FFFF 1 ST4A #stack(gr1,-4)=gr31
0| 000004 stw 93C1FFFF 1 ST4A #stack(gr1,-8)=gr30
0| 000008 stwu 9421FF70 1 ST4U gr1,#stack(gr1,-144)=gr1
0| 00000C lwz 83C20004 1 L4A gr30=+CONSTANT_AREA(gr2,0)
>> 0| 000010 stfd D84100A8 1 STDPL price(gr1,168)=fp2
>> 0| 000014 stfd D86100B0 1 STDPL price(gr1,176)=fp3
9| 000018 mffs FC00048E 1 LFFSCR fp0=fcf
9| 00001C stfd D8010078 1 STPL #MX_SET1(gr1,120)=fp0
9| 000020 lwz 80010078 1 L4A gr0=#MX_SET1(gr1,120)
9| 000024 rlwinm 5400077E 1 RN4 gr0=gr0,0,0x7
9| 000028 stw 90010070 1 ST4A original_rounding_mode(gr1,112)=gr0
10| 00002C mtfscfi FF81410C 1 SETDRND fcf=4,fcf
11| 000030 lfd C80100A8 1 LDPL fp0=price(gr1,168)
11| 000034 lfd C82100B0 1 LDPL fp1=price(gr1,176)
11| 000038 lfd C85E0000 1 LDPL fp2=+CONSTANT_AREA(gr30,0)
11| 00003C lfd C87E0008 1 LDPL fp3=+CONSTANT_AREA(gr30,8)
11| 000040 dmulq FC001044 1 MDPE fp0,fp1=fp0-fp3,fcf
11| 000044 stfdp F4010040 1 STDFE gst(gr1,64)=fp0,fp1
12| 000048 lfd C80100A8 1 LDPL fp0=price(gr1,168)
12| 00004C lfd C82100B0 1 LDPL fp1=price(gr1,176)
12| 000050 lfd C85E0010 1 LDPL fp2=+CONSTANT_AREA(gr30,16)
12| 000054 lfd C87E0018 1 LDPL fp3=+CONSTANT_AREA(gr30,24)
12| 000058 dmulq FC001044 1 MDPE fp0,fp1=fp0-fp3,fcf
12| 00005C stfdp F4010050 1 STDFE pst(gr1,80)=fp0,fp1
13| 000060 lfd C80100A8 1 LDPL fp0=price(gr1,168)
13| 000064 lfd C82100B0 1 LDPL fp1=price(gr1,176)
13| 000068 lfd C8410040 1 LDPL fp2=gst(gr1,64)
13| 00006C lfd C8610048 1 LDPL fp3=gst(gr1,72)
13| 000070 daddq FC001004 1 ADPE fp0,fp1=fp0-fp3,fcf

```

```

13 000074 lfd C8410050 1 LDFL fp2=pst(gr1,80)
13 000078 lfd C8610058 1 LDFL fp3=pst(gr1,88)
13 00007C daddq FC001004 1 ADFF fp0,fp1=fp0-fp3,fcf
13 000080 stfdp F4010060 1 STDFE total(gr1,96)=fp0,fp1
14 000084 lwz 80010070 1 L4A gr0=original_rounding_mode(gr1,112)
14 000088 stw 90010078 1 ST4A #MX_SET1(gr1,120)=gr0
14 00008C lfd C8010078 1 LFL fp0=#MX_SET1(gr1,120)
14 000090 mtfsf FC03058E 1 LFSCR8 fsr,fcf=fp0,1,1
>> 15 000094 lfd C8410060 1 LDFL fp2=total(gr1,96)
>> 15 000098 lfd C8610068 1 LDFL fp3=total(gr1,104)
16 CL.1:
16 00009C lwz 83C10088 1 L4A gr30=#stack(gr1,136)
16 0000A0 addi 38210090 1 AI gr1=gr1,144
16 0000A4 bclr 4E800020 1 BA lr

```

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Special registers in the PowerPC®:

The Special-Purpose Register Conventions shows that PowerPC® special purpose registers (SPRs).

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 4. Special-Purpose Register Conventions

Register or Register Field	Status	Use
LR	volatile	Used as a branch target address or holds a return address.
CTR	volatile	Used for loop count decrement and branching.
XER	volatile	Fixed-point exception register.
FPSCR	volatile	Floating-point exception register.
CR0, CR1	volatile	Condition-register bits.
CR2, CR3, CR4	nonvolatile	Condition-register bits.
CR5, CR6, CR7	volatile	Condition-register bits.

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.

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Runtime process stack:

The stack format convention is designed to enhance the efficiency of Prolog and epilog function usage, parameter passing and the shared library support.

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.

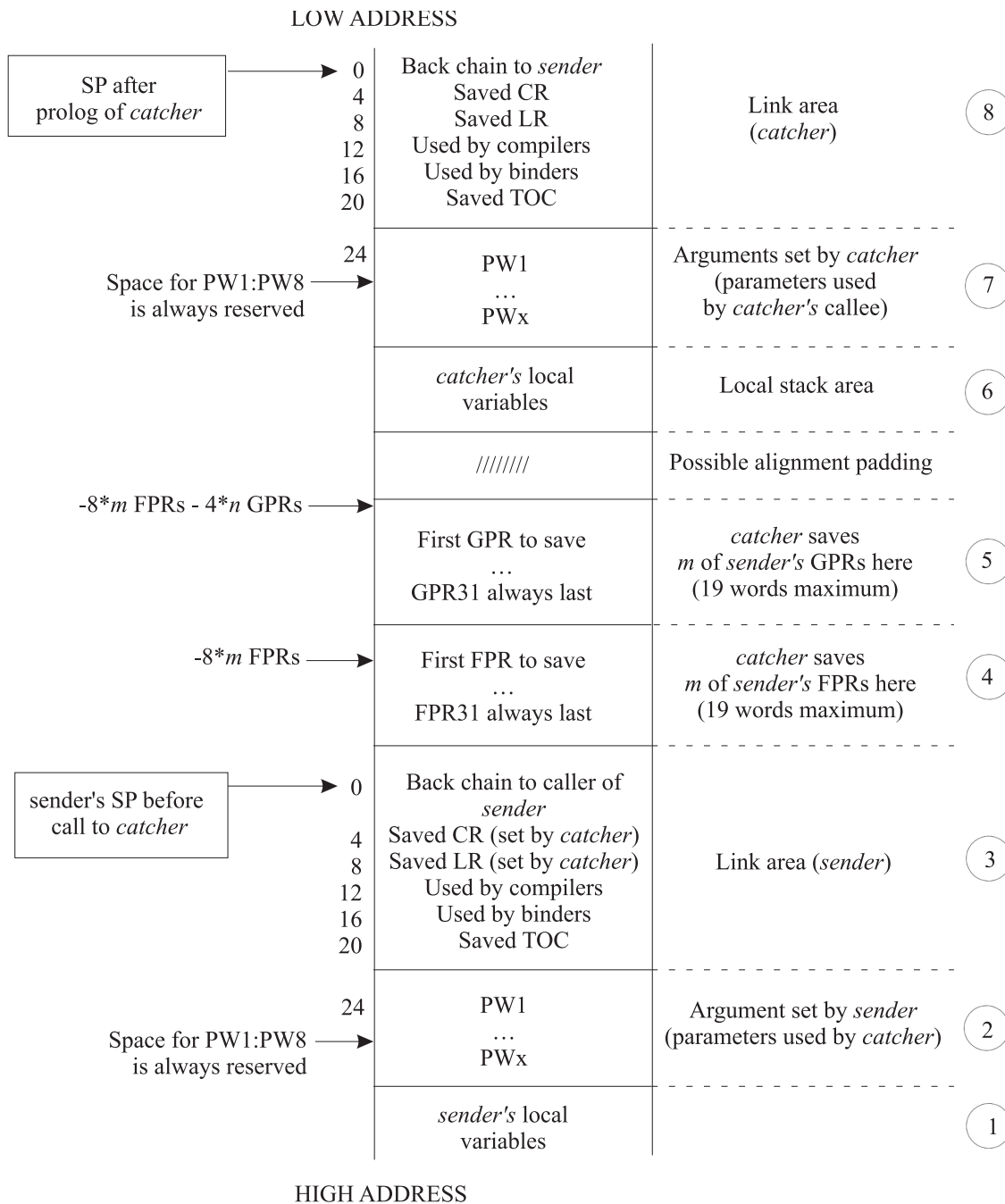


Figure 2. Runtime Stack

Stack layout:

The stack layout grows from numerically higher storage addresses to numerically lower addresses.

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.

Note:

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.)

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Stack areas:

The stack layout is divided into eight areas numbered 1 to 8, starting from the bottom of the diagram to the top of the diagram.

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 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).

Note:

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:

$$\begin{aligned}
 &18 \times 8 \\
 &(\text{for FPRs}) \\
 &+ 19 \times 4 \\
 &(\text{for GPRs}) \\
 &= 220
 \end{aligned}$$

- **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).

Related concepts:

"Assembling and linking a program" on page 73

The assembly language program defines the commands for assembling and linking a program.

"Interpreting an assembler listing" on page 77

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

"Interpreting a symbol cross-reference" on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

"Subroutine linkage convention" on page 83

The Subroutine Linkage Convention.

"Understanding and programming the toc" on page 110

The TOC is used to find objects in an XCOFF file.

"Running a program" on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

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.

Related concepts:

"Assembling and linking a program" on page 73

The assembly language program defines the commands for assembling and linking a program.

"Interpreting an assembler listing" on page 77

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

"Interpreting a symbol cross-reference" on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

"Subroutine linkage convention" on page 83

The Subroutine Linkage Convention.

"Understanding and programming the toc" on page 110

The TOC is used to find objects in an XCOFF file.

"Running a program" on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Prologs and epilogs:

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

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.

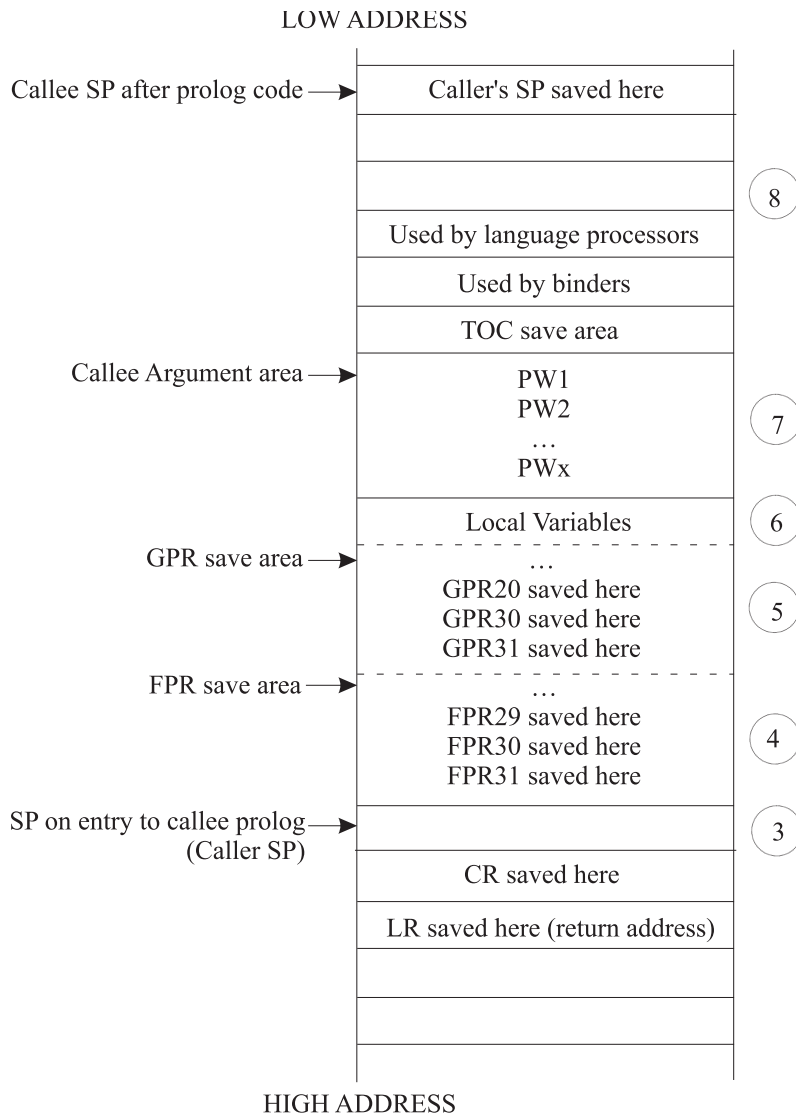


Figure 3. Stack Frame Layout

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

If:	Then:
Any nonvolatile FPRs (FPR14:FPR31) are used	Save them in the FPR save area (area 4 in the previous figure).
Any nonvolatile GPRs (GPR13:GPR31) are used	Save them in the GPR save area (area 5 in the previous figure).
LR is used for a nonleaf procedure	Save the LR at offset eight from the caller function SP.
Any of the nonvolatile condition register (CR) fields are used.	Save the CR at offset four from the caller function SP.
A new stack frame is required	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.

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

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

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 a 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.

Note:

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.

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

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.

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.

```
#Following are the prolog/epilog of a function that saves n GPRS #(n>2):
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 R!>    #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:

```
_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 beings 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:

```

_restgpr0_13    lwz    r13,-76(r1)          #restore r13
_restgpr0_14    lwz    r14,-72(r1)          #restore r14
_restgpr0_15    lwz    r15,-68(r1)          #restore r15
_restgpr0_16    lwz    r16,-64(r1)          #restore r16
_restgpr0_17    lwz    r17,-60(r1)          #restore r17
_restgpr0_18    lwz    r18,-56(r1)          #restore r18
_restgpr0_19    lwz    r19,-52(r1)          #restore r19
_restgpr0_20    lwz    r20,-48(r1)          #restore r20
_restgpr0_21    lwz    r21,-44(r1)          #restore r21
_restgpr0_22    lwz    r22,-40(r1)          #restore r22
_restgpr0_23    lwz    r23,-36(r1)          #restore r23
_restgpr0_24    lwz    r24,-32(r1)          #restore r24
_restgpr0_25    lwz    r25,-28(r1)          #restore r25
_restgpr0_26    lwz    r26,-24(r1)          #restore r26
_restgpr0_27    lwz    r27,-20(r1)          #restore r27
_restgpr0_28    lwz    r28,-16(r1)          #restore r28
_restgpr0_29    lwz    r0,8(r1)             #get return
                                           #address from
                                           #frame
                lwz    r29,-12(r1)          #restore r29
                mtlr   r0                   #move return
                                           #address to LR
                lwz    r30,-8(r1)           #restore r30
                lwz    r31,-4(r1)           #restore r31
                blr                          #return

```

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

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

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.

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:

```
#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 **alloca** subroutine to dynamically allocate stack space.

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

```
_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
_savegpr1_21    stw    r21,-44(r12)      #save r21
_savegpr1_22    stw    r22,-40(r12)      #save r22
_savegpr1_23    stw    r23,-36(r12)      #save r23
_savegpr1_24    stw    r24,-32(r12)      #save r24
_savegpr1_25    stw    r25,-28(r12)      #save r25
_savegpr1_26    stw    r26,-24(r12)      #save r26
_savegpr1_27    stw    r27,-20(r12)      #save r27
_savegpr1_28    stw    r28,-16(r12)      #save r28
_savegpr1_29    stw    r29,-12(r12)      #save r29
              stw    r30,-8(r12)         #save r30
              stw    r31,-4(r12)         #save r31
              blr                      #return
```

The following example shows an FPR save routine:


```

_savefpr_14    stfd    f14,-144(r1)      #save f14
_savefpr_15    stfd    f15,-136(r1)      #save f15
_savefpr_16    stfd    f16,-128(r1)      #save f16
_savefpr_17    stfd    f17,-120(r1)      #save f17
_savefpr_18    stfd    f18,-112(r1)      #save f18
_savefpr_19    stfd    f19,-104(r1)      #save f19
_savefpr_20    stfd    f20,-96(r1)       #save f20
_savefpr_21    stfd    f21,-88(r1)       #save f21
_savefpr_22    stfd    f22,-80(r1)       #save f22
_savefpr_23    stfd    f23,-72(r1)       #save f23
_savefpr_24    stfd    f24,-64(r1)       #save f24
_savefpr_25    stfd    f25,-56(r1)       #save f25
_savefpr_26    stfd    f26,-48(r1)       #save f26
_savefpr_27    stfd    f27,-40(r1)       #save f27
_savefpr_28    stfd    f28,-32(r1)       #save f28
_savefpr_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 a GPR restore routine when FPRs are saved:

```

_restgpr1_13   lwz     r13,-76(r12)      #restore r13
_restgpr1_14   lwz     r14,-72(r12)      #restore r14
_restgpr1_15   lwz     r15,-68(r12)      #restore r15
_restgpr1_16   lwz     r16,-64(r12)      #restore r16
_restgpr1_17   lwz     r17,-60(r12)      #restore r17
_restgpr1_18   lwz     r18,-56(r12)      #restore r18
_restgpr1_19   lwz     r19,-52(r12)      #restore r19
_restgpr1_20   lwz     r20,-48(r12)      #restore r20
_restgpr1_21   lwz     r21,-44(r12)      #restore r21
_restgpr1_22   lwz     r22,-40(r12)      #restore r22
_restgpr1_23   lwz     r23,-36(r12)      #restore r23
_restgpr1_24   lwz     r24,-32(r12)      #restore r24
_restgpr1_25   lwz     r25,-28(r12)      #restore r25
_restgpr1_26   lwz     r26,-24(r12)      #restore r26
_restgpr1_27   lwz     r27,-20(r12)      #restore r27
_restgpr1_28   lwz     r28,-16(r12)      #restore r28
_restgpr1_29   lwz     r29,-12(r12)      #restore r29
               lwz     r30,-8(r12)       #restore r30
               lwz     r31,-4(r12)       #restore r31
               blr                     #return

```

The following example shows an FPR restore routine:

```

_restfpr_14    lfd     r14,-144(r1)      #restore r14
_restfpr_15    lfd     r15,-136(r1)      #restore r15
_restfpr_16    lfd     r16,-128(r1)      #restore r16
_restfpr_17    lfd     r17,-120(r1)      #restore r17
_restfpr_18    lfd     r18,-112(r1)      #restore r18
_restfpr_19    lfd     r19,-104(r1)      #restore r19
_restfpr_20    lfd     r20,-96(r1)       #restore r20
_restfpr_21    lfd     r21,-88(r1)       #restore r21
_restfpr_22    lfd     r22,-80(r1)       #restore r22
_restfpr_23    lfd     r23,-72(r1)       #restore r23
_restfpr_24    lfd     r24,-64(r1)       #restore r24
_restfpr_25    lfd     r25,-56(r1)       #restore r25
_restfpr_26    lfd     r26,-48(r1)       #restore r26
_restfpr_27    lfd     r27,-40(r1)       #restore r27
_restfpr_28    lfd     r28,-32(r1)       #restore r28
_restfpr_29    lwz     r0,8(r1)          #get return
                                   #address from
                                   #frame
               lfd     r29,-24(r1)       #restore r29
               mtlr    r0               #move return
                                   #address to LR

```

```

lfd    r30,-16(r1)      #restore r30
lfd    r31,-8(r1)       #restore r31
blr                    #return

```

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

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.

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:

```

#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

```

Note:

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 `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.

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Updating the stack pointer:

The PowerPC® **stwu** (Store Word with Update) instruction is used for computing the new SP and saving the back chain.

The PowerPC® **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:

```
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
stwu     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:

```
stwu     r1, <-frame_size>(r1)    #update SP and save caller's SP
```

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

Calling routine's responsibilities

An assembler language program calls another program, the caller should not use the names of the called programs commands, functions, or procedures as global assembler language symbols.

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** file) 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.

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Called routine's responsibilities

Prologs and epilogs are used in the called routines.

Prologs and epilogs 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.

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Traceback tags

The assembly (compiled) program needs traceback information for the debugger to examine if the program traps or crashes during the execution.

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 fullword 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.

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Example

Example of assembler code called by a C subroutine.

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

```
#      Call this assembly routine from C routine:
#      callfile.c:
#      main()
#      {
#      examlinkage();
#      }
#      Compile as follows:
#      cc -o callfile callfile.c examlinkage.s
#
#####
#      On entry to a procedure(callee), all or some of the
#      following steps should be done:
#      1. Save the link register at offset 8 from the
#         stack pointer for non-leaf procedures.
#      2. If any of the CR bits 8-19(CR2,CR3,CR4) is used
#         then save the CR at displacement 4 of the current
#         stack pointer.
#      3. Save all non-volatile FPRs used by this routine.
#         If more than three non-volatile FPR are saved,
#         a call to ._savefn can be used to
#         save them (n is the number of the first FPR to be
#         saved).
#      4. Save all non-volatile GPRs used by this routine
#         in the caller's GPR SAVE area (negative displacement
#         from the current stack pointer r1).
#      5. Store back chain and decrement stack pointer by the
#         size of the stack frame.
#
#      On exit from a procedure (callee), all or some of the
#      following steps should be done:
#      1. Restore all GPRs saved.
#      2. Restore stack pointer to value it had on entry.
#      3. Restore Link Register if this is a non-leaf
#         procedure.
#      4. Restore bits 20-31 of the CR if it was saved.
#      5. Restore all FPRs saved. If any FPRs were saved then
#         a call to ._savefn can be used to restore them
#         (n is the first FPR to be restored).
#      6. Return to caller.
#####
#      The following routine calls printf() to print a string.
#      The routine performs entry steps 1-5 and exit steps 1-6.
#      The prolog/epilog code is for small stack frame size.
#      DSA + 8 < 32k
#####
.file "examlinkage.s"
#Static data entry in T(able)O(f)C(ontents)
.toc
T.examlkage.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 T(able)O(f)C(ontents)
.toc
T.examlkage: .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.
mfcrr   12
#       Save FPRs 14-31.
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)
#####
#load static data address
#####
        l       14,T.examlinkage.c(2)
#       Load string address which is an argument to printf.
cal      3, printing(14)
#       Call to printf routine
bl       .printf[PR]
cror     31, 31, 31
#EPILOG: Return Sequence
#       Restore stack ptr
ai       1, 1, szdsa
#       Restore GPRs 13-31.
lm       13, -8*nfprs-4*ngprs(1)
#       Restore FPRs 14-31.
bl       ._restf14
cror     31, 31, 31
#       Get saved LR.
l        0, 8(1)
#       Get saved CR if this routine saved it.
l        12, 4(1)
#       Move return address to link register.
mtlrr    0
#       Restore CR2, CR3, & CR4 of the CR.
mtcrf    0x38,12
#       Return to address held in Link Register.
brl
        .tbttag 0x0,0xc,0x0,0x0,0x0,0x0,0x0,0x0
#       External variables
        .extern ._savef14
        .extern ._restf14
        .extern .printf[PR]
#####
#       Data
#####
        .csect  examlinkage.c[rw]
        .align  2
printing: .byte  'E','x','a','m','p','l','e',' ','f','o','r','
        .byte  'P','R','I','N','T','I','N','G'
        .byte  0xa,0x0

```

Using milicode routines

The milicode routines contain machine-dependent and performance-critical functions.

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.

Note:

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:

Item	Description
__mulh	Calculates the high-order 32 bits of the integer product $arg1 * arg2$. Input R3 = $arg1$ (signed integer) R4 = $arg2$ (signed integer) Output R3 = high-order 32 bits of $arg1*arg2$ POWER® family Register Usage GPR3, GPR4, MQ PowerPC® Register Usage GPR3, GPR4
__mull	Calculates 64 bits of the integer product $arg1 * arg2$, returned in two 32-bit registers. Input R3 = $arg1$ (signed integer) R4 = $arg2$ (signed integer) Output R3 = high-order 32 bits of $arg1*arg2$ R4 = low-order 32 bits of $arg1*arg2$ 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 $arg1/arg2$. For division by zero and overflow, the quotient and remainder are undefined and may vary by implementation. Input R3 = $arg1$ (dividend) (signed integer) R4 = $arg2$ (divisor) (signed integer) Output R3 = quotient of $arg1/arg2$ (signed integer) R4 = remainder of $arg1/arg2$ (signed integer) POWER® family Register Usage GPR3, GPR4, MQ PowerPC® Register Usage GPR0, GPR3, GPR4

Item	Description
__divus	Calculated the 32-bit quotient and 32-bit remainder of unsigned integers <i>arg1/arg2</i>. For division by zero and overflow, the quotient and remainder are undefined and may vary by implementation. Input R3 = <i>arg1</i> (dividend) (unsigned integer) R4 = <i>arg2</i> (divisor) (unsigned integer) Output R3 = quotient of <i>arg1/arg2</i> (unsigned integer) R4 = remainder of <i>arg1/arg2</i> (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 <i>arg1/arg2</i>. For division by zero and overflow, the quotient and remainder are undefined and may vary by implementation. Input R3 = <i>arg1</i> (dividend) (signed integer) R4 = <i>arg2</i> (divisor) (signed integer) Output R3 = quotient of <i>arg1/arg2</i> (signed integer) POWER® family Register Usage GPR3, GPR4, MQ PowerPC® Register Usage GPR3, GPR4
__quous	Calculates the 32-bit quotient of unsigned integers <i>arg1/arg2</i>. For division by zero and overflow, the quotient and remainder are undefined and may vary by implementation. Input R3 = <i>arg1</i> (dividend) (unsigned integer) R4 = <i>arg2</i> (divisor) (unsigned integer) Output R3 = quotient of <i>arg1/arg2</i> (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 **mulh** milicode routine in an assembler program:

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

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Understanding assembler passes” on page 75

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

“b (Branch) instruction” on page 176

“cror (Condition Register OR) instruction” on page 202

Understanding and programming the toc

The TOC is used to find objects in an XCOFF file.

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:

Item	Description
.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 $(0...(n-1))*4$ to the offset, where n is the position of the csect address defined with the “.tc pseudo-op” on page 558.

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Using the toc

The TOC is created with certain conventions.

To use the TOC, you must follow certain conventions:

- General-Purpose Register 2 always contains a pointer to the TOC.
- All references from the **.text** section of an assembler program to **.data** or the **.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 **.text** and **.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.

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

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.

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 **.toc** and **.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.

...
l 5,TCA(RTOC)               #Now GPR5 contains the
                               #address of a[rw].
...

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

```

Accessing data through the toc entry with the te storage-mapping class

The TE storage-mapping class is used to access external data.

As is the case with the TC storage-mapping class, the TE storage-mapping class can be used to access external data. An external data item is accessed by first loading that item's address from the TOC, and then using that address to get the data. To avoid the generation of TOC-overflow code, the TE symbol is loaded from the TOC with a two-instruction sequence as shown in the following example:

```

.toc
.tc      a[TE],a[RW]

.extern a[RW]
.csect  prog1[PR]
...
addis   3,a[TE](2)          # R_TOCU relocation used by default.
ld      5,a[TE](3)          # R_TOCL relocation used by default.
# Now GPR5 contains the address of a[RW]

```

The two instructions to load a[TE] from the TOC do not have to be sequential, but adding an offset to the referenced symbol is not allowed. For example, the ld instruction cannot be as follows:

```
ld      5,a[TE]+8(3)        # Invalid reference
```

The selection of the storage-mapping class and the R_TOCU and R_TOCL relocation types can be selected independently. For example, a[TE] can be used as a normal TOC symbol with the following instruction:

```
ld      5,a[TE]@tc(2)       # GPR5 contains the address of a[RW]
```

The two-instruction sequence can also be used with a[TC] from the previous example:

```

addis   5,a[TC]@u(2)
ld      5,a[TC]@l(5)        # GPR5 contains the address of a[RW]

```

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.

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:

The source for the main C program td1.c

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

```
.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:

The source for the C main program td2.c

1. The following is the source for the C main program 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);
}
```

2. The following is the assembler source for 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)
br
.toc
.comm    t_data[TD],4 # t_data is a global symbol

```

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:

Example of 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

The data section is accessed through TOC using a feature that allows intermodule calls to be used.

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 run time, either from the system or from the dynamic load of another object file. See the **ld** command for information about import lists.

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

```
.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.
```

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

```
#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.
        1      2,stkto(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:  1      12,prog2(2)      #Get address of
                                #out-of-module
                                #descriptor.
        st      2,stkto(1)      #save callers' toc.
        1      0,0(12)         #Get its entry address
                                #from descriptor.
        1      2,4(12)         #Get its toc from
                                #descriptor.
        mtctr    0              #Put into Count Register.
        bctr     0              #Return to entry address
                                #in Count Register.
                                #Return is directly to
                                #original caller.
```

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Using thread local storage

Thread-local variables can be declared and defined with the TL and UL storage-mapping classes.

Thread-local variables can be declared and defined with the TL and UL storage-mapping classes. The thread-local variables are referenced using code sequences defined by the AIX implementation.

A thread-local variable has a region handle and a region offset. In general, the `__tls_get_addr()` function is called to compute the address of a thread-local variable for the calling thread, given the variables region handle and offset.

Other access methods can be used in more restricted circumstances. The **local-exec** access method is used by the main program to reference variables also defined in the main program. The **initial-exec** access method is used to reference thread-local variables defined in the main program or any shared object loaded along with the main program. The **local-dynamic** access method is used by a module to reference thread-local variables defined in the same module.

Access to thread-local storage makes use of routines in the pthread library that have nonstandard calling conventions. These routines are `__tls_get_addr()` and `__tls_get_mod()`. The routine used in 32-bit programs is `__get_tpoffset()`. In 64-bit programs, the current thread pointer is always contained in **gpr13**.

An uninitialized thread-local storage variable *bar* can be declared with the following statement:

```
.comm bar[UL]
```

Similarly, a thread-local, initialized, integer variable *foo* can be declared with the following statements:

```
.csect foo[TL]
.long      1
```

Access method	32-bit code	64-bit code(if different)	Comment
General-dynamic access method	.tc foo[TC],foo[TL]		Variable offset
	.tc .foo[TC],foo[TL]@m		Region handle
	lwz 4,foo[TC](2)	ld 4,foo[TC](2)	
	lwz 3,.foo[TC](2)	ld 3,.foo[TC](2)	
	bla __tls_get_addr		Modifies r0,r3,r4,r5,r11,lr,cr0
	#r3 = &foo		

Access method	32-bit code	64-bit code(if different)	Comment
Local-dynamic access method	.tc foo[TC],foo[TL]@ld		Variable offset, ld relocation specifier
	.tc mh[TC],mh[TC]@ml		Module handle for the caller
	lwz 3,mh[TC](2)	ld 3,mh[TC](2)	
	bla __tls_get_mod		Modifies r0,r3,r4,r5,r11,lr,cr0
	#r3 = &TLS for module		
	lwz 4,foo[TC](2)	ld 4,foo[TC](2)	
	add 5,3,4		Compute <i>&foo</i>
	.rename mh[TC], "_\$TLSML"		Symbol for the module handle must have the name "_\$TLSML"
Initial-exec access method	.tc foo[TC],foo[TL]@ie		Variable offset, ie relocation specifier
	bla __get_tpointer()	# r13 contains tpointer	__get_tpointer modifies r3
	lwz 4,foo[TC](2)	ld 4,foo[TC](2)	
	add 5,3,4	add 5,4,13	Compute <i>&foo</i>
Local-exec access method	.tc foo[TC],foo[TL]@le		Variable offset, le relocation specifier
	bla __get_tpointer()	# r13 contains tpointer	__get_tpointer modifies r3
	lwz 4,foo[TC](2)	ld 4,foo[TC](2)	Compute <i>&foo</i>
	add 5,3,4	add 5,4,13	

The local-dynamic and local-exec access methods have a faster code sequence that can be used if the total size of thread-local variables is smaller than 62 KB. If the total size of the region is too large, the link-editor will patch the code by generating extra instructions, negating the benefit of using the faster code sequence.

Access method	32-bit code	Comment
Local-dynamic access method	.tc mh[TC],mh[TC]@ml	Module handle for the caller
	.rename mh[TC], "_\$TLSML"	Symbol for the module handle must have the name "_\$TLSML"
	lwz 3,mh[TC](2)	
	bla __tls_get_mod	
	la 4,foo[TL]@ld(3)	r4 = <i>&foo</i>
Local-exec access method	bla __get_tpointer	Modifies r3
	la 4,foo[TL]@ld(3)	r4 = <i>&foo</i>
	# OR	
	lwz 5,foo[TL]@le(13)	r5 = <i>foo</i>

Access method	64-bit code	Comment
Local-dynamic access method	.tc mh[TC],mh[TC]@ml	Module handle for the caller
	.rename mh[TC], "_\$TLSML"	Symbol for the module handle must have the name "_\$TLSML"
	ld 3,mh[TC](2)	
	bla __tls_get_mod	
	la 4,foo[TL]@ld(3)	r4 = <i>&foo</i>
Local-exec access method	la 4,foo[TL]@le(13)	r4 = <i>&foo</i>
	# OR	
	lwz 5,foo[TL]@le(13)	r5 = <i>foo</i>

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program”

A program is ready to run when it has been assembled and linked without producing any error messages.

Running a program

A program is ready to run when it has been assembled and linked without producing any error messages.

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 concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

Related information:

as

dbx

Extended instruction mnemonics

The assembler supports a set of extended mnemonics and symbols to simplify assembly language programming.

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

The assembler supports extended mnemonics for different types of Register 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.

Related concepts:

“Extended mnemonics of fixed-point load instructions” on page 129

The extended mnemonics for fixed-point load instructions for POWER® family and PowerPC®.

Branch mnemonics that incorporate only the bo operand

The instruction format for extended mnemonics that incorporate only the *BO* field.

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)

Mnemonics	Input Operands	Equivalent to	
bdz, bdza, bdzl, bdzla	<i>target_addr</i>	bc, bca, bcl, bcla	18, 0, <i>target_addr</i>
bdn, bdna, bdnl, bdnla	<i>target_addr</i>	bc, bca, bcl, bcla	16, 0, <i>target_addr</i>
bdzr, bdzrl	None	bcr, bcrl	18, 0
bdnr, bdnrl	None	bcr, bcrl	16, 0
bbt, bbta, bbtl, bbtla	1) <i>BI, target_addr</i>	bc, bca, bcl, bcla	12, <i>BI, target_addr</i>
	2) <i>target_addr</i>		12, 0, <i>target_addr</i>
bbf, bbfa, bbfl, bbfla	1) <i>BI, target_addr</i>	bc, bca, bcl, bcla	4, <i>BI, target_addr</i>
	2) <i>target_addr</i>		4, 0, <i>target_addr</i>
bbtr, bbtc, bbtrl, bbtcl	1) <i>BI</i>	bcr, bcc, bcrl, bccl	12, <i>BI</i>
	2) None		12, 0
bbfr, bbfc, bbfrl, bbfcrl	1) <i>BI</i>	bcr, bcc, bcrl, bccl	4, <i>BI</i>
	2) None		4, 0
br, bctr, brl, bctrl	None	bcr, bcc, bcrl, bccl	20, 0

Table 8. PowerPC® Extended Mnemonics (BO Field Only)

Mnemonics	Input Operands	Equivalent to
bdz, bdza, bdzl, bdzla	<i>target_addr</i>	bc, bca, bcl, bcla 18, 0, <i>target_addr</i>
bdnz, bdnza, bdnzl, bdnzla	<i>target_addr</i>	bc, bca, bcl, bcla 16, 0, <i>target_addr</i>
bdzlr, bdzlr	None	bclr, bclrl 18, 0
bdnzlr, bdnzlr	None	bclr, bclrl 16, 0
bt, bta, btl, btl	1) <i>BI, target_addr</i>	bc, bca, bcl, bcla 12, <i>BI, target_addr</i>
	2) <i>target_addr</i>	12, 0, <i>target_addr</i>
bf, bfa, bfl, bfl	1) <i>BI, target_addr</i>	bc, bca, bcl, bcla 4, <i>BI, target_addr</i>
	2) <i>target_addr</i>	4, 0, <i>target_addr</i>
bdzt, bdzta, bdztl, bdztla	1) <i>BI, target_addr</i>	bc, bca, bcl, bcla 10, <i>BI, target_addr</i>
	2) <i>target_addr</i>	10, 0, <i>target_addr</i>
bdzf, bdzfa, bdzfl, bdzfla	1) <i>BI, target_addr</i>	bc, bca, bcl, bcla 2, <i>BI, target_addr</i>
	2) <i>target_addr</i>	2, 0, <i>target_addr</i>
bdnzt, bdnzta, bdnztl, bdnztla	1) <i>BI, target_addr</i>	bc, bca, bcl, bcla 8, <i>BI, target_addr</i>
	2) <i>target_addr</i>	8, 0, <i>target_addr</i>
bdnzf, bdnzfa, bdnzfl, bdnzfla	1) <i>BI, target_addr</i>	bc, bca, bcl, bcla 0, <i>BI, target_addr</i>
	2) <i>target_addr</i>	0, 0, <i>target_addr</i>
btlr, btctr, btlr, btctrl	1) <i>BI</i>	bclr, bcctr, bclrl, bcctrl 12, <i>BI</i>
	2) None	12, 0
bflr, bfctr, bflr, bfctrl	1) <i>BI</i>	bclr, bcctr, bclrl, bcctrl 4, <i>BI</i>
	2) None	4, 0
bdztlr, bdztlr	1) <i>BI</i>	bclr, bclrl 10, <i>BI</i>
	2) None	10, 0
bdzflr, bdzflr	1) <i>BI</i>	bclr, bclrl 2, <i>BI</i>
	2) None	2, 0
bdnztlr, bdnztlr	1) <i>BI</i>	bclr, bclrl 8, <i>BI</i>
	2) None	8, 0
bdnzflr, bdnzflr	1) <i>BI</i>	bclr, bclrl 0, <i>BI</i>
	2) None	0, 0
blr, bctr, blrl, bctrl	None	bclr, bcctr, bclrl, bcctrl 20, 0

Related concepts:

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER® family and PowerPC®.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

“Differences between POWER® family and PowerPC® instructions with the same op code” on page 145

The differences between POWER® family and PowerPC® instructions with the same op code

“POWER® family instructions deleted from PowerPC®” on page 151

The POWER® family instructions deleted from PowerPC®.

“Added PowerPC® instructions” on page 152

The instructions that have been added to PowerPC®, but are not in the POWER® family.

“Instructions available only for the PowerPC® 601 RISC microprocessor” on page 152

The PowerPC® instructions that are available only for the PowerPC® 601 RISC Microprocessor.

Extended branch mnemonics that incorporate the bo field and a partial bi field

The extended branch mnemonics instruction format when the *BO* field and *BI* field are incorporated.

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:

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

The assembler supports six encoding values for the *BO* operand:

- Branch if condition true (*BO*=12):

POWER® family	PowerPC®
bxx	bxx
bxxa	bxxa
bxxl	bxxl
bxxla	bxxla
bxxr	bxxlr
bxxrl	bxxlrl
bxxc	bxxctr
bxxcl	bxxctrl

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

- Branch if condition false (*BO*=04):

POWER® family	PowerPC®
bxx	bxx
bxxa	bxxa
bxxl	bxxl
bxxla	bxxla
bxxr	bxxlr
bxxrl	bxxlrl
bxxc	bxxctr
bxxcl	bxxctrl

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):

- **bdnxx**

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):

- **bdnxx**

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):

- **bdzxx**

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):

- **bdzxx**

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 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:

Bit	Description
0	Less than; floating-point less than
1	Greater than; floating-point greater than
2	Equal; floating-point equal
3	Summary overflow; floating-point unordered

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 9. *BI* Operand Symbols for Basic and Extended Branch Conditional Mnemonics

Symbol	Value	Meaning
lt	0	less than
gt	1	greater than
eq	2	equal
so	3	summary overflow
un	3	unordered (after floating-point comparison)
cr0	0	CR field 0
cr1	1	CR field 1
cr2	2	CR field 2
cr3	3	CR field 3
cr4	4	CR field 4
cr5	5	CR field 5
cr6	6	CR field 6
cr7	7	CR field 7

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:

```
bdnzt 4*cr5+eq, target_addr
```

This is equivalent to:

```
bc 8, 22, target_addr
```

2. To decrement CTR, then branch only if CTR is not zero and condition in CR0 is equal:

```
bdnzt eq, target_addr
```

This is equivalent to:

```
bc 8, 2, 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:

```
bdz target_addr
```

This is equivalent to:

```
bc 18, 0, 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:

```
bge target_addr
```

This is equivalent to:

```
bc 4, 0, target_addr
```

2. To branch to an absolute target if CR4 indicates greater than, and set the Link register:

```
bgtla cr4, target_addr
```

This is equivalent to:

```
bcla 12, 17, 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.

Related concepts:

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER® family and PowerPC®.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

“Differences between POWER® family and PowerPC® instructions with the same op code” on page 145

The differences between POWER® family and PowerPC® instructions with the same op code

“POWER® family instructions deleted from PowerPC®” on page 151

The POWER® family instructions deleted from PowerPC®.

“Added PowerPC® instructions” on page 152

The instructions that have been added to PowerPC®, but are not in the POWER® family.

“Instructions available only for the PowerPC® 601 RISC microprocessor” on page 152

The PowerPC® instructions that are available only for the PowerPC® 601 RISC Microprocessor.

Extended mnemonics for branch prediction

The assembler source program can have information on branch conditional instruction by adding a branch prediction suffix to the mnemonic of the instruction.

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:

Item	Description
+	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-	bcctrl+	bcctrl-
bcl+	bcl-	bcla+	bcla-
bclr+	bclr-	bclr1+	bclr1-
bdneq+	bdneq-	bdnge+	bdnge-
bdngt+	bdngt-	bdnle+	bdnle-
bdnlt+	bdnlt-	bdnne+	bdnne-
bdnns+	bdnns-	bdnso+	bdnso-
bdnz+	bdnz-	bdnza+	bdnza-
bdnzf+	bdnzf-	bdnzfa+	bdnzfa-
bdnzfl+	bdnzfl-	bdnzfla+	bdnzfla-

Assembler language reference **125**

bso+	bso-	bsoa+	bsoa-
bsoctr+	bsoctr-	bsoctrl+	bsoctrl-
bsol+	bsol-	bsola+	bsola-
bsolr+	bsolr-	bsolrl+	bsolrl-
bt+	bt-	bta+	bta-
btctr+	btctr-	btctrl+	btctrl-
btl+	btl-	btla+	btla-
btlr+	btlr-	btlrl+	btlrl-
bun+	bun-	buna+	buna-
bunctr+	bunctr-	bunctrl+	bunctrl-
bunl+	bunl-	bunla+	bunla-
bunlr+	bunlr-	bunlrl+	bunlrl-
bz+	bz-	bza+	bza-
bzctr+	bzctr-	bzctrl+	bzctrl-
bzl+	bzl-	bzla+	bzla-
bzlr+	bzlr-	bzlr+	bzlr-

Related concepts:

“Extended instruction mnemonics” on page 118

The assembler supports a set of extended mnemonics and symbols to simplify assembly language programming.

“Extended mnemonics of condition register logical instructions”

The extended mnemonics of condition register logical instructions are available in POWER[®] family and PowerPC[®].

“Extended mnemonics of fixed-point arithmetic instructions” on page 127

The extended mnemonics of fixed-point arithmetic instructions for POWER[®] family and PowerPC[®].

“Extended mnemonics of fixed-point compare instructions” on page 128

The extended mnemonics for fixed-point compare instructions.

“Extended mnemonics of fixed-point load instructions” on page 129

The extended mnemonics for fixed-point load instructions for POWER[®] family and PowerPC[®].

“Extended mnemonics of moving from or to special-purpose registers” on page 132

Extended mnemonics of moving from or to special- purpose registers.

“Extended mnemonics of 32-bit fixed-point rotate and shift instructions” on page 137

The extended mnemonics of 32-bit fixed-point rotate and shift instructions.

“.bc pseudo-op” on page 518

“bclr or bcr (Branch Conditional Link Register) instruction” on page 182

Extended mnemonics of condition register logical instructions

The extended mnemonics of condition register logical instructions are available in POWER[®] family and PowerPC[®].

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 10. Condition Register Logical Instruction Extended Mnemonics

Extended Mnemonic	Equivalent to	Meaning
crset <i>bx</i>	creqv <i>bx, bx, bx</i>	Condition register set
crclr <i>bx</i>	crxor <i>bx, bx, bx</i>	Condition register clear
crmove <i>bx, by</i>	cror <i>bx, by, by</i>	Condition register move
crnot <i>bx, by</i>	crnor <i>bx, by, by</i>	Condition register NOT

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:

```
crxor    3, 3, 3
```

2. To clear the EQ bit of CR3:

```
crclr    4*cr3+eq
```

This is equivalent to:

```
crxor    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 concepts:

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER[®] family and PowerPC[®].

“Functional differences for POWER[®] family and PowerPC[®] instructions” on page 144

The POWER[®] family and PowerPC[®] instructions that share the same op code on POWER[®] family and PowerPC[®] platforms, but differ in their functional definition.

“Differences between POWER[®] family and PowerPC[®] instructions with the same op code” on page 145

The differences between POWER[®] family and PowerPC[®] instructions with the same op code

“POWER[®] family instructions deleted from PowerPC[®]” on page 151

The POWER[®] family instructions deleted from PowerPC[®].

“Added PowerPC[®] instructions” on page 152

The instructions that have been added to PowerPC[®], but are not in the POWER[®] family.

“Instructions available only for the PowerPC[®] 601 RISC microprocessor” on page 152

The PowerPC[®] instructions that are available only for the PowerPC[®] 601 RISC Microprocessor.

Extended mnemonics of fixed-point arithmetic instructions

The extended mnemonics of fixed-point arithmetic instructions for POWER[®] family and PowerPC[®].

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 11. Fixed-Point Arithmetic Instruction Extended Mnemonics

Extended Mnemonic	Equivalent to	Meaning
subi <i>rx, ry, value</i>	addi <i>rx, ry, -value</i>	Subtract Immediate
subis <i>rx, ry, value</i>	addis <i>rx, ry, -value</i>	Subtract Immediate Shifted
subic[.] <i>rx, ry, value</i>	addic[.] <i>rx, ry, -value</i>	Subtract Immediate
subc[o][.] <i>rx, ry, rz</i>	subfc[o][.] <i>rx, rz, ry</i>	Subtract
si[.] <i>rt, ra, value</i>	ai[.] <i>rt, ra, -value</i>	Subtract Immediate
sub[o][.] <i>rx, ry, rz</i>	subf[o][.] <i>rx, rz, ry</i>	Subtract

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

Related concepts:

“Extended mnemonics of fixed-point logical instructions” on page 130

The extended mnemonics of fixed-point logical instructions.

“Extended instruction mnemonics” on page 118

The assembler supports a set of extended mnemonics and symbols to simplify assembly language programming.

“Extended mnemonics of branch instructions” on page 119

The assembler supports extended mnemonics for different types of Register instructions.

“Extended mnemonics of condition register logical instructions” on page 126

The extended mnemonics of condition register logical instructions are available in POWER® family and PowerPC®.

“Extended mnemonics of fixed-point compare instructions”

The extended mnemonics for fixed-point compare instructions.

“Extended mnemonics of fixed-point load instructions” on page 129

The extended mnemonics for fixed-point load instructions for POWER® family and PowerPC®.

“Extended mnemonics of fixed-point trap instructions” on page 131

The extended mnemonics for fixed-point trap instructions incorporate the most useful TO operand values.

“Extended mnemonics of moving from or to special-purpose registers” on page 132

Extended mnemonics of moving from or to special- purpose registers.

“Extended mnemonics of 32-bit fixed-point rotate and shift instructions” on page 137

The extended mnemonics of 32-bit fixed-point rotate and shift instructions.

Extended mnemonics of fixed-point compare instructions

The extended mnemonics for 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

Extended Mnemonic	Equivalent to	Meaning
cmpdi <i>ra, value</i>	cmpi 0, 1, <i>ra, value</i>	Compare Word Immediate
cmpwi <i>bf, ra, si</i>	cmpi <i>bf, 0, ra, si</i>	Compare Word Immediate
cmpd <i>ra, rb</i>	cmp 0, 1, <i>ra, rb</i>	Compare Word
cmpw <i>bf, ra, rb</i>	cmp <i>bf, 0, ra, rb</i>	Compare Word
cmpldi <i>ra, value</i>	cmpli 0, 1, <i>ra, value</i>	Compare Logical Word Immediate
cmplwi <i>bf, ra, ui</i>	cmpli <i>bf, 0, ra, ui</i>	Compare Logical Word Immediate
cmpld <i>ra, rb</i>	cmpl 0, 1, <i>ra, rb</i>	Compare Logical Word
cmplw <i>bf, ra, rb</i>	cmpl <i>bf, 0, ra, rb</i>	Compare Logical Word

Related concepts:

“Extended instruction mnemonics” on page 118

The assembler supports a set of extended mnemonics and symbols to simplify assembly language programming.

“Extended mnemonics of branch instructions” on page 119

The assembler supports extended mnemonics for different types of Register instructions.

“Extended mnemonics of condition register logical instructions” on page 126

The extended mnemonics of condition register logical instructions are available in POWER® family and PowerPC®.

“Extended mnemonics of fixed-point compare instructions” on page 128

The extended mnemonics for fixed-point compare instructions.

“Extended mnemonics of fixed-point load instructions”

The extended mnemonics for fixed-point load instructions for POWER® family and PowerPC®.

“Extended mnemonics of fixed-point trap instructions” on page 131

The extended mnemonics for fixed-point trap instructions incorporate the most useful TO operand values.

“Extended mnemonics of moving from or to special-purpose registers” on page 132

Extended mnemonics of moving from or to special- purpose registers.

“Extended mnemonics of 32-bit fixed-point rotate and shift instructions” on page 137

The extended mnemonics of 32-bit fixed-point rotate and shift instructions.

Extended mnemonics of fixed-point load instructions

The extended mnemonics for fixed-point load instructions for POWER® family and PowerPC®.

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

Extended Mnemonic	Equivalent to	Meaning
li <i>rx, value</i>	addi <i>rx, 0, value</i>	Load Immediate
la <i>rx, disp(ry)</i>	addi <i>rx, ry, disp</i>	Load Address
lil <i>rt, value</i>	cal <i>rt, value(0)</i>	Load Immediate Lower
liu <i>rt, value</i>	cau <i>rt, 0, value</i>	Load Immediate Upper
lis <i>rx, value</i>	addis <i>rx, 0, value</i>	Load Immediate Shifted

Related concepts:

“Extended instruction mnemonics” on page 118

The assembler supports a set of extended mnemonics and symbols to simplify assembly language programming.

“Extended mnemonics of branch instructions” on page 119

The assembler supports extended mnemonics for different types of Register instructions.

“Extended mnemonics of condition register logical instructions” on page 126

The extended mnemonics of condition register logical instructions are available in POWER® family and PowerPC®.

“Extended mnemonics of fixed-point arithmetic instructions” on page 127

The extended mnemonics of fixed-point arithmetic instructions for POWER® family and PowerPC®.

“Extended mnemonics of fixed-point logical instructions”

The extended mnemonics of fixed-point logical instructions.

“Extended mnemonics of fixed-point trap instructions” on page 131

The extended mnemonics for fixed-point trap instructions incorporate the most useful TO operand values.

“Extended mnemonics of moving from or to special-purpose registers” on page 132

Extended mnemonics of moving from or to special- purpose registers.

“Extended mnemonics of 32-bit fixed-point rotate and shift instructions” on page 137

The extended mnemonics of 32-bit fixed-point rotate and shift instructions.

“addi (Add Immediate) or cal (Compute Address Lower) instruction” on page 162

Extended mnemonics of fixed-point logical instructions

The 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 **com** assembly mode.

Table 14. Fixed-Point Logical Instruction Extended Mnemonics

Extended Mnemonic	Equivalent to	Meaning
nop	ori 0, 0, 0	OR Immediate
mr[.] rx,ry	or[.] rx, ry, ry	OR
notl[.] rx,ry	nor[.] rx, ry, ry	NOR

Related concepts:

“Extended instruction mnemonics” on page 118

The assembler supports a set of extended mnemonics and symbols to simplify assembly language programming.

“Extended mnemonics of branch instructions” on page 119

The assembler supports extended mnemonics for different types of Register instructions.

“Extended mnemonics of fixed-point compare instructions” on page 128

The extended mnemonics for fixed-point compare instructions.

“Extended mnemonics of fixed-point load instructions” on page 129

The extended mnemonics for fixed-point load instructions for POWER® family and PowerPC®.

“Extended mnemonics of fixed-point trap instructions” on page 131

The extended mnemonics for fixed-point trap instructions incorporate the most useful TO operand values.

“Extended mnemonics of moving from or to special-purpose registers” on page 132

Extended mnemonics of moving from or to special- purpose registers.

“Extended mnemonics of 32-bit fixed-point rotate and shift instructions” on page 137

The extended mnemonics of 32-bit fixed-point rotate and shift instructions.

“nor (NOR) instruction” on page 377

“or (OR) instruction” on page 378

“ori or oril (OR Immediate) instruction” on page 381

Extended mnemonics of fixed-point trap instructions

The extended mnemonics for fixed-point trap instructions incorporate the most useful TO operand values.

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 **com** assembly mode.

Table 15. Fixed-Point Trap Instruction Codes

Code	TO Encoding	Meaning
lt	10000	less than
le	10100	less than or equal
ng	10100	not greater than
eq	00100	equal
ge	01100	greater than or equal
nl	01100	not less than
gt	01000	greater than
ne	11000	not equal
llt	00010	logically less than
lle	00110	logically less than or equal
lng	00110	logically not greater than
lge	00101	logically greater than or equal
lnl	00101	logically not less than
lgt	00001	logically greater than
lne	00011	logically not equal
None	11111	Unconditional

The POWER[®] family extended mnemonics for fixed-point trap instructions have the following format:

- **txx** or **txxi**

where *xx* is one of the codes specified in the preceding table.

The 64-bit PowerPC[®] extended mnemonics for doubleword, fixed-point trap instructions have the following format:

- **tdxx** or **tdxxi**

The PowerPC[®] extended mnemonics for fixed-point trap instructions have the following formats:

- **twxx** or **twxxi**

where *xx* is one of the codes specified in the preceding table.

The **trap** instruction is an unconditional trap:

- **trap**

Examples

1. To trap if R10 is less than R20:

```
tlt    10, 20
```

This is equivalent to:

```
t      16, 10, 20
```

2. To trap if R4 is equal to 0x10:

```
teqi    4, 0x10
```

This is equivalent to:

```
ti    0x4, 4, 0x10
```

3. To trap unconditionally:

```
trap
```

This is equivalent to:

```
tw    31, 0, 0
```

4. To trap if RX is not equal to RY:

```
twnei   RX, 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
```

Extended mnemonic mtr for moving to the condition register

The **mtr** (Move to Condition Register) extended mnemonic copies the contents of the low order 32 bits of a general purpose register (GPR).

The **mtr** (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 **mfcrr** instruction.

The extended mnemonic **mtr** *Rx* is equivalent to the instruction **mtrcf** 0xFF,*Rx*.

This extended mnemonic is in the **com** assembly mode.

Extended mnemonics of moving from or to special-purpose registers

Extended mnemonics of moving from or to special- purpose registers.

This article discusses the following extended mnemonics:

mf spr extended mnemonics for POWER® family

mf spr Extended Mnemonics for POWER® family

Table 16. *mf spr* Extended Mnemonics for POWER® family

Extended Mnemonic	Equivalent to	Privileged	SPR Name
mf xer <i>rt</i>	mf spr <i>rt</i> ,1	no	XER
mf lrr <i>rt</i>	mf spr <i>rt</i> ,8	no	LR
mf ctr <i>rt</i>	mf spr <i>rt</i> ,9	no	CTR
mf mq <i>rt</i>	mf spr <i>rt</i> ,0	no	MQ
mf rtcu <i>rt</i>	mf spr <i>rt</i> ,4	no	RTCU
mf rtcl <i>rt</i>	mf spr <i>rt</i> ,5	no	RTCL
mf dec <i>rt</i>	mf spr <i>rt</i> ,6	no	DEC
mf tid <i>rt</i>	mf spr <i>rt</i> ,17	yes	TID
mf dsisr <i>rt</i>	mf spr <i>rt</i> ,18	yes	DSISR

Table 16. *mfspr* Extended Mnemonics for POWER® family (continued)

Extended Mnemonic	Equivalent to	Privileged	SPR Name
mfdar <i>rt</i>	mfspr <i>rt,19</i>	yes	DAR
mfsdr0 <i>rt</i>	mfspr <i>rt,24</i>	yes	SDR0
mfsdr1 <i>rt</i>	mfspr <i>rt,25</i>	yes	SDR1
mfsrr0 <i>rt</i>	mfspr <i>rt,26</i>	yes	SRR0
mfsrr1 <i>rt</i>	mfspr <i>rt,27</i>	yes	SRR1

Related concepts:

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER® family and PowerPC®.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

“Differences between POWER® family and PowerPC® instructions with the same op code” on page 145

The differences between POWER® family and PowerPC® instructions with the same op code

“POWER® family instructions deleted from PowerPC®” on page 151

The POWER® family instructions deleted from PowerPC®.

“Added PowerPC® instructions” on page 152

The instructions that have been added to PowerPC®, but are not in the POWER® family.

“Instructions available only for the PowerPC® 601 RISC microprocessor” on page 152

The PowerPC® instructions that are available only for the PowerPC® 601 RISC Microprocessor.

mtspr extended mnemonics for POWER® family

mtspr Extended Mnemonics for POWER® family

Table 17. *mtspr* Extended Mnemonics for POWER® family

Extended Mnemonic	Equivalent to	Privileged	SPR Name
mfxr <i>rs</i>	mtspr <i>1,rs</i>	no	XER
mflr <i>rs</i>	mtspr <i>8,rs</i>	no	LR
mtctr <i>rs</i>	mtspr <i>9,rs</i>	no	CTR
mtmq <i>rs</i>	mtspr <i>0,rs</i>	no	MQ
mtrtcu <i>rs</i>	mtspr <i>20,rs</i>	yes	RTCU
mtrtcl <i>rs</i>	mtspr <i>21,rs</i>	yes	RTCL
mtdec <i>rs</i>	mtspr <i>22,rs</i>	yes	DEC
mttid <i>rs</i>	mtspr <i>17,rs</i>	yes	TID
mtdsisr <i>rs</i>	mtspr <i>18,rs</i>	yes	DSISR
mtdar <i>rs</i>	mtspr <i>19,rs</i>	yes	DAR
mtsdr0 <i>rs</i>	mtspr <i>24,rs</i>	yes	SDR0
mtsdr1 <i>rs</i>	mtspr <i>25,rs</i>	yes	SDR1
mtsrr0 <i>rs</i>	mtspr <i>26,rs</i>	yes	SRR0
mtsrr1 <i>rs</i>	mtspr <i>27,rs</i>	yes	SRR1

Related concepts:

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER® family and PowerPC®.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

“Differences between POWER® family and PowerPC® instructions with the same op code” on page 145
The differences between POWER® family and PowerPC® instructions with the same op code
“POWER® family instructions deleted from PowerPC®” on page 151
The POWER® family instructions deleted from PowerPC®.

“Added PowerPC® instructions” on page 152

The instructions that have been added to PowerPC®, but are not in the POWER® family.

“Instructions available only for the PowerPC® 601 RISC microprocessor” on page 152

The PowerPC® instructions that are available only for the PowerPC® 601 RISC Microprocessor.

mf spr extended mnemonics for PowerPC®

mf spr Extended Mnemonics for PowerPC®

Table 18. mf spr Extended Mnemonics for PowerPC®

Extended Mnemonic	Equivalent to	Privileged	SPR Name
mf xer rt	mf spr rt,1	no	XER
mf l r rt	mf spr rt,8	no	LR
mf ctr rt	mf spr rt,9	no	CTR
mf dsisr rt	mf spr rt,18	yes	DSISR
mf dar rt	mf spr rt,19	yes	DAR
mf dec rt	mf spr rt,22	yes	DEC
mf sdr1 rt	mf spr rt,25	yes	SDR1
mf srr0 rt	mf spr rt,26	yes	SRR0
mf srr1 rt	mf spr rt,27	yes	SRR1
mf sprg rt,0	mf spr rt,272	yes	SPRG0
mf sprg rt,1	mf spr rt,273	yes	SPRG1
mf sprg rt,2	mf spr rt,274	yes	SPRG2
mf sprg rt,3	mf spr rt,275	yes	SPRG3
mf ear rt	mf spr rt,282	yes	EAR
mf pvr rt	mf spr rt,287	yes	PVR
mf ibatu rt,0	mf spr rt,528	yes	IBAT0U
mf ibatl rt,1	mf spr rt,529	yes	IBAT0L
mf ibatu rt,1	mf spr rt,530	yes	IBAT1U
mf ibatl rt,1	mf spr rt,531	yes	IBAT1L
mf ibatu rt,2	mf spr rt,532	yes	IBAT2U
mf ibatl rt,2	mf spr rt,533	yes	IBAT2L
mf ibatu rt,3	mf spr rt,534	yes	IBAT3U
mf ibatl rt,3	mf spr rt,535	yes	IBAT3L
mf dbatu rt,0	mf spr rt,536	yes	DBAT0U
mf dbatl rt,0	mf spr rt,537	yes	DBAT0L
mf dbatu rt,1	mf spr rt,538	yes	DBAT1U
mf dbatl rt,1	mf spr rt,539	yes	DBAT1L
mf dbatu rt,2	mf spr rt,540	yes	DBAT2U
mf dbatl rt,2	mf spr rt,541	yes	DBAT2L
mf dbatu rt,3	mf spr rt,542	yes	DBAT3U
mf dbatl rt,3	mf spr rt,543	yes	DBAT3L

Note: The **mf dec** instruction is a privileged instruction in PowerPC®. The encoding for this instruction in PowerPC® differs from that in POWER® family. See the **mf spr** (Move from Special-Purpose Register)

Instruction for information on this instruction. Differences between POWER® family and PowerPC® Instructions with the Same Op Code provides a summary of the differences for this instruction for POWER® family and PowerPC®.

mtspr extended mnemonics for PowerPC®

mtspr Extended Mnemonics for PowerPC®

Table 19. mtspr Extended Mnemonics for PowerPC®

Extended Mnemonic	Equivalent to	Privileged	SPR Name
mtxer <i>rs</i>	mtspr 1, <i>rs</i>	no	XER
mtlir <i>rs</i>	mtspr 8, <i>rs</i>	no	LR
mtctr <i>rs</i>	mtspr 9, <i>rs</i>	no	CTR
mtdsisr <i>rs</i>	mtspr 19, <i>rs</i>	yes	DSISR
mtdar <i>rs</i>	mtspr 19, <i>rs</i>	yes	DAR
mtdec <i>rs</i>	mtspr 22, <i>rs</i>	yes	DEC
mtsdr1 <i>rs</i>	mtspr 25, <i>rs</i>	yes	SDR1
mtsrr0 <i>rs</i>	mtspr 26, <i>rs</i>	yes	SRR0
mtsrr1 <i>rs</i>	mtspr 27, <i>rs</i>	yes	SRR1
mtsprg 0, <i>rs</i>	mtspr 272, <i>rs</i>	yes	SPRG0
mtsprg 1, <i>rs</i>	mtspr 273, <i>rs</i>	yes	SPRG1
mtsprg 2, <i>rs</i>	mtspr 274, <i>rs</i>	yes	SPRG2
mtsprg 3, <i>rs</i>	mtspr 275, <i>rs</i>	yes	SPRG3
mtear <i>rs</i>	mtspr 282, <i>rs</i>	yes	EAR
mttbl <i>rs</i> (or mttb <i>rs</i>)	mtspr 284, <i>rs</i>	yes	TBL
mttbu <i>rs</i>	mtspr 285, <i>rs</i>	yes	TBU
mtibatu 0, <i>rs</i>	mtspr 528, <i>rs</i>	yes	IBAT0U
mtibatl 0, <i>rs</i>	mtspr 529, <i>rs</i>	yes	IBAT0L
mtibatu 1, <i>rs</i>	mtspr 530, <i>rs</i>	yes	IBAT1U
mtibatl 1, <i>rs</i>	mtspr 531, <i>rs</i>	yes	IBAT1L
mtibatu 2, <i>rs</i>	mtspr 532, <i>rs</i>	yes	IBAT2U
mtibatl 2, <i>rs</i>	mtspr 533, <i>rs</i>	yes	IBAT2L
mtibatu 3, <i>rs</i>	mtspr 534, <i>rs</i>	yes	IBAT3U
mtibatl 3, <i>rs</i>	mtspr 535, <i>rs</i>	yes	IBAT3L
mtdbatu 0, <i>rs</i>	mtspr 536, <i>rs</i>	yes	DBAT0U
mtdbatl 0, <i>rs</i>	mtspr 537, <i>rs</i>	yes	DBAT0L
mtdbatu 1, <i>rs</i>	mtspr 538, <i>rs</i>	yes	DBAT1U
mtdbatl 1, <i>rs</i>	mtspr 539, <i>rs</i>	yes	DBAT1L
mtdbatu 2, <i>rs</i>	mtspr 540, <i>rs</i>	yes	DBAT2U
mtdbatl 2, <i>rs</i>	mtspr 541, <i>rs</i>	yes	DBAT2L
mtdbatu 3, <i>rs</i>	mtspr 542, <i>rs</i>	yes	DBAT3U
mtdbatl 3, <i>rs</i>	mtspr 543, <i>rs</i>	yes	DBAT3L

Note: The **mfdec** instruction is a privileged instruction in PowerPC®. The encoding for this instruction in PowerPC® differs from that in POWER® family.

mfspir extended mnemonics for PowerPC® 601 RISC microprocessor

mfspir Extended Mnemonics for PowerPC® 601 RISC Microprocessor

Table 20. *mf spr* Extended Mnemonics for PowerPC® 601 RISC Microprocessor

Extended Mnemonic	Equivalent to	Privileged	SPR Name
mfmq <i>rt</i>	mf spr <i>rt</i> ,0	no	MQ
mf xer <i>rt</i>	mf spr <i>rt</i> ,1	no	XER
mf rt cu <i>rt</i>	mf spr <i>rt</i> ,4	no	RTCU
mf rt cl <i>rt</i>	mf spr <i>rt</i> ,5	no	RTCL
mf dec <i>rt</i>	mf spr <i>rt</i> ,6	no	DEC
mf lr <i>rt</i>	mf spr <i>rt</i> ,8	no	LR
mf ctr <i>rt</i>	mf spr <i>rt</i> ,9	no	CTR
mf dsisr <i>rt</i>	mf spr <i>rt</i> ,18	yes	DSISR
mf dar <i>rt</i>	mf spr <i>rt</i> ,19	yes	DAR
mf sdr1 <i>rt</i>	mf spr <i>rt</i> ,25	yes	SDR1
mf srr0 <i>rt</i>	mf spr <i>rt</i> ,26	yes	SRR0
mf srr1 <i>rt</i>	mf spr <i>rt</i> ,27	yes	SRR1
mf sprg <i>rt</i> ,0	mf spr <i>rt</i> ,272	yes	SPRG0
mf sprg <i>rt</i> ,1	mf spr <i>rt</i> ,273	yes	SPRG1
mf sprg <i>rt</i> ,2	mf spr <i>rt</i> ,274	yes	SPRG2
mf sprg <i>rt</i> ,3	mf spr <i>rt</i> ,275	yes	SPRG3
mf ear <i>rt</i>	mf spr <i>rt</i> ,282	yes	EAR
mf pvr <i>rt</i>	mf spr <i>rt</i> ,287	yes	PVR

mtspr extended mnemonics for PowerPC® 601 RISC microprocessor

mtspr Extended Mnemonics for PowerPC® 601 RISC Microprocessor

Table 21. *mtspr* Extended Mnemonics for PowerPC® 601 RISC Microprocessor

Extended Mnemonic	Equivalent to	Privileged	SPR Name
mtmq <i>rs</i>	mtspr 0, <i>rs</i>	no	MQ
mt xer <i>rs</i>	mtspr 1, <i>rs</i>	no	XER
mt lr <i>rs</i>	mtspr 8, <i>rs</i>	no	LR
mt ctr <i>rs</i>	mtspr 9, <i>rs</i>	no	CTR
mt dsisr <i>rs</i>	mtspr 18, <i>rs</i>	yes	DSISR
mt dar <i>rs</i>	mtspr 19, <i>rs</i>	yes	DAR
mt rt cu <i>rs</i>	mtspr 20, <i>rs</i>	yes	RTCU
mt rt cl <i>rs</i>	mtspr 21, <i>rs</i>	yes	RTCL
mt dec <i>rs</i>	mtspr 22, <i>rs</i>	yes	DEC
mt sdr1 <i>rs</i>	mtspr 25, <i>rs</i>	yes	SDR1
mt srr0 <i>rs</i>	mtspr 26, <i>rs</i>	yes	SRR0
mt srr1 <i>rs</i>	mtspr 27, <i>rs</i>	yes	SRR1
mt sprg 0, <i>rs</i>	mtspr 272, <i>rs</i>	yes	SPRG0
mt sprg 1, <i>rs</i>	mtspr 273, <i>rs</i>	yes	SPRG1
mt sprg 2, <i>rs</i>	mtspr 274, <i>rs</i>	yes	SPRG2
mt sprg 3, <i>rs</i>	mtspr 275, <i>rs</i>	yes	SPRG3
mt ear <i>rs</i>	mtspr 282, <i>rs</i>	yes	EAR

Related concepts:

“Migrating source programs” on page 143

The assembler issues errors and warnings if a source program contains instructions that are not in the current assembly mode.

“Functional differences for POWER[®] family and PowerPC[®] instructions” on page 144

The POWER[®] family and PowerPC[®] instructions that share the same op code on POWER[®] family and PowerPC[®] platforms, but differ in their functional definition.

“Differences between POWER[®] family and PowerPC[®] instructions with the same op code” on page 145

The differences between POWER[®] family and PowerPC[®] instructions with the same op code

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER[®] family and PowerPC[®].

“POWER[®] family instructions deleted from PowerPC[®]” on page 151

The POWER[®] family instructions deleted from PowerPC[®].

“Added PowerPC[®] instructions” on page 152

The instructions that have been added to PowerPC[®], but are not in the POWER[®] family.

Extended mnemonics of 32-bit fixed-point rotate and shift instructions

The 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:

Related concepts:

“Extended instruction mnemonics” on page 118

The assembler supports a set of extended mnemonics and symbols to simplify assembly language programming.

“Extended mnemonics of branch instructions” on page 119

The assembler supports extended mnemonics for different types of Register instructions.

“Extended mnemonics of fixed-point arithmetic instructions” on page 127

The extended mnemonics of fixed-point arithmetic instructions for POWER[®] family and PowerPC[®].

“Extended mnemonics of condition register logical instructions” on page 126

The extended mnemonics of condition register logical instructions are available in POWER[®] family and PowerPC[®].

“Extended mnemonics of fixed-point compare instructions” on page 128

The extended mnemonics for fixed-point compare instructions.

“Extended mnemonics of fixed-point logical instructions” on page 130

The extended mnemonics of fixed-point logical instructions.

“Extended mnemonics of moving from or to special-purpose registers” on page 132

Extended mnemonics of moving from or to special- purpose registers.

“bcctr or bcc (Branch Conditional to Count Register) instruction” on page 179

“addi (Add Immediate) or cal (Compute Address Lower) instruction” on page 162

“twi or ti (Trap Word Immediate) instruction” on page 508

Alternative input format

The alternative input format for POWER[®] family and PowerPC[®] instructions.

The alternative input format is applied to the following POWER[®] family and PowerPC[®] instructions.

POWER® family

rlimi[.]
 rlinm[.]
 rlnm[.]
 rlmi[.]

PowerPC®

rlwimi[.]
 rlwinm[.]
 rlwnm[.]
 Not applicable

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:

Examples of valid 32-bit masks.

The following shows examples of valid 32-bit masks.

		0	15	31
MB = 0	ME = 31	11111111111111111111111111111111		
MB = 0	ME = 0	10000000000000000000000000000000		
MB = 0	ME = 22	1111111111111111111111110000000000		
MB = 12	ME = 25	00000000000111111111111100000000		
MB = 22	ME = 31	00000000000000000000000111111111		
MB = 29	ME = 6	11111100000000000000000000000111		

Examples of 32-bit masks that are not valid:

Examples of 32-bit masks that are not valid.

The following shows examples of 32-bit masks that are not valid.

0	15	31
00000000000000000000000000000000		
01010101010101010101010101010101		
00000000000111100000110000000000		
11111100000111111111111100000000		

Related concepts:

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER® family and PowerPC®.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

“Differences between POWER® family and PowerPC® instructions with the same op code” on page 145

The differences between POWER® family and PowerPC® instructions with the same op code

“POWER[®] family instructions deleted from PowerPC[®]” on page 151

The POWER[®] family instructions deleted from PowerPC[®].

“Added PowerPC[®] instructions” on page 152

The instructions that have been added to PowerPC[®], but are not in the POWER[®] family.

“Instructions available only for the PowerPC[®] 601 RISC microprocessor” on page 152

The PowerPC[®] instructions that are available only for the PowerPC[®] 601 RISC Microprocessor.

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.

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:

Item	Description
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 22. 32-bit Rotate and Shift Extended Mnemonics for PowerPC[®]

Operation	Extended Mnemonic	Equivalent to	Restrictions
Extract and left justify immediate	extlwi RA, RS, n, b	rlwinm $RA, RS, b, 0, n-1$	$32 > n > 0$
Extract and right justify immediate	extrwi RA, RS, n, b	rlwinm $RA, RS, b+n, 32-n, 31$	$32 > n > 0 \ \& \ b+n \leq 32$
Insert from left immediate	inslwi RA, RS, n, b	rlwinm $RA, RS, 32-b, b, (b+n)-1$	$b+n \leq 32 \ \& \ 32 > n > 0 \ \& \ 32 > b \geq 0$
Insert from right immediate	insrwi RA, RS, n, b	rlwinm $RA, RS, 32-(b+n), b, (b+n)-1$	$b+n \leq 32 \ \& \ 32 > n > 0$
Rotate left immediate	rotlwi RA, RS, n	rlwinm $RA, RS, n, 0, 31$	$32 > n \geq 0$
Rotate right immediate	rotrwi RA, RS, n	rlwinm $RA, RS, 32-n, 0, 31$	$32 > n \geq 0$
Rotate left	rotlw RA, RS, b	rlwinm $RA, RS, RB, 0, 31$	None
Shift left immediate	slwi RA, RS, n	rlwinm $RA, RS, n, 0, 31-n$	$32 > n \geq 0$
Shift right immediate	srwi RA, RS, n	rlwinm $RA, RS, 32-n, n, 31$	$32 > n \geq 0$
Clear left immediate	clrlwi RA, RS, n	rlwinm $RA, RS, 0, n, 31$	$32 > n \geq 0$

Table 22. 32-bit Rotate and Shift Extended Mnemonics for PowerPC® (continued)

Operation	Extended Mnemonic	Equivalent to	Restrictions
Clear right immediate	clrrwi <i>RA, RS, n</i>	rlwinm <i>RA, RS, 0, 0, 31-n</i>	$32 > n \geq 0$
Clear left and shift left immediate	clrlwi <i>RA, RS, b, n</i>	rlwinm <i>RA, RS, b-n, 31-n</i>	$b-n \geq 0 \ \& \ 32 > n \geq 0 \ \& \ 32 > b \geq 0$

Note:

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

Example of 32-bit Rotate and Shift Extended Mnemonics for POWER® family and PowerPC®

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, RY, 16
```

This is equivalent to:

```
rlwinm    RX, RY, 0, 16, 31
```

Extended mnemonics of 64-bit fixed-point rotate and shift instructions

The 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
- Extended mnemonics of 32-bit fixed-point rotate and shift instructions

Related concepts:

“Extended instruction mnemonics” on page 118

The assembler supports a set of extended mnemonics and symbols to simplify assembly language programming.

“Extended mnemonics of branch instructions” on page 119

The assembler supports extended mnemonics for different types of Register instructions.

“Extended mnemonics of fixed-point compare instructions” on page 128

The extended mnemonics for fixed-point compare instructions.

“Extended mnemonics of fixed-point logical instructions” on page 130

The extended mnemonics of fixed-point logical instructions.

“Extended mnemonics of fixed-point trap instructions” on page 131

The extended mnemonics for fixed-point trap instructions incorporate the most useful TO operand values.

“Extended mnemonics of moving from or to special-purpose registers” on page 132

Extended mnemonics of moving from or to special- purpose registers.

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER[®] family and PowerPC[®].

“Functional differences for POWER[®] family and PowerPC[®] instructions” on page 144

The POWER[®] family and PowerPC[®] instructions that share the same op code on POWER[®] family and PowerPC[®] platforms, but differ in their functional definition.

“POWER[®] family instructions deleted from PowerPC[®]” on page 151

The POWER[®] family instructions deleted from PowerPC[®].

“Added PowerPC[®] instructions” on page 152

The instructions that have been added to PowerPC[®], but are not in the POWER[®] family.

“Instructions available only for the PowerPC[®] 601 RISC microprocessor” on page 152

The PowerPC[®] instructions that are available only for the PowerPC[®] 601 RISC Microprocessor.

Alternative input format

The alternative input format applied to POWER[®] family and PowerPC[®] instructions.

The alternative input format is applied to the following POWER[®] family and PowerPC[®] instructions.

POWER[®] family

rlimi[.]

rlinm[.]

rlnm[.]

rlmi[.]

PowerPC[®]

rlwimi[.]

rlwinm[.]

rlwnm[.]

Not applicable

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 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:

Item	Description
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®

Operation	Extended Mnemonic	Equivalent to	Restrictions
Extract double word and right justify immediate	extrdi RA, RS, n, b	rldicl $RA, RS, b + n, 64 - n$	$n > 0$
Rotate double word left immediate	rotldi RA, RS, n	rldicl $RA, RS, n, 0$	None
Rotate double word right immediate	rotrdi RA, RS, n	rldicl $RA, RS, 64 - n, 0$	None
Rotate double word right immediate	srdi RA, RS, n	rldicl $RA, RS, 64 - n, n$	$n < 64$
Clear left double word immediate	clrldi RA, RS, n	rldicl $RA, RS, 0, n$	$n < 64$
Extract double word and left justify immediate	extldi RA, RS, n, b	rldicr $RA, RS, b, n - 1$	None
Shift left double word immediate	sldi RA, RS, n	rldicr $RA, RS, n, 63 - n$	None
Clear right double word immediate	clrrdi RA, RS, n	rldicr $RA, RS, 0, 63 - n$	None
Clear left double word and shift left immediate	clrslldi RA, RS, b, n	rldic $RA, RS, n, b - n$	None
Insert double word from right immediate	insrdi RA, RS, n, b	rldimi $RA, RS, 64 - (b + n), b$	None
Rotate double word left	rotld RA, RS, RB	rldcl $RA, RS, RB, 0$	None

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 concepts:

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER® family and PowerPC®.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and

PowerPC® platforms, but differ in their functional definition.

“Differences between POWER® family and PowerPC® instructions with the same op code” on page 145

The differences between POWER® family and PowerPC® instructions with the same op code

“POWER® family instructions deleted from PowerPC®” on page 151

The POWER® family instructions deleted from PowerPC®.

“Added PowerPC® instructions” on page 152

The instructions that have been added to PowerPC®, but are not in the POWER® family.

“Instructions available only for the PowerPC® 601 RISC microprocessor” on page 152

The PowerPC® instructions that are available only for the PowerPC® 601 RISC Microprocessor.

Migrating source programs

The assembler issues errors and warnings if a source program contains instructions that are not in the current assembly mode.

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 **lswx** instruction for these errors. The **lswx** instruction requires that the registers specified by the second and third operands (*RA* and *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 **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** for more information on invalid instruction forms.

If the **mfspr** and **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 **read_real_time** and **time_base_to_time** routines instead of code accessing the real time clock or time base SPRs.)

Related concepts:

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER® family and PowerPC®.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

“Differences between POWER® family and PowerPC® instructions with the same op code” on page 145

The differences between POWER® family and PowerPC® instructions with the same op code

“POWER® family instructions deleted from PowerPC®” on page 151

The POWER® family instructions deleted from PowerPC®.

“Added PowerPC® instructions” on page 152

The instructions that have been added to PowerPC®, but are not in the POWER® family.

“Instructions available only for the PowerPC® 601 RISC microprocessor” on page 152
The PowerPC® instructions that are available only for the PowerPC® 601 RISC Microprocessor.

Functional differences for POWER® family and PowerPC® instructions

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

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

POWER® family	PowerPC®	Description
dcs	sync	The sync instruction causes more pervasive synchronization in PowerPC® than the dcs instruction does in POWER® family.
ics	isync	The isync instruction causes more pervasive synchronization in PowerPC® than the ics instruction does in POWER® family.
svca	sc	In POWER® family, information from MSR is saved into CTR. In PowerPC®, this information is saved into SRR1. PowerPC® 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.
mtsri	mtsrin	POWER® family uses the RA field to compute the segment register number and, in some cases, the effective address (EA) is stored. PowerPC® has no RA field, and the EA is not stored.
lsx	lswx	POWER® family does not alter the target register <i>RT</i> if the string length is 0. PowerPC® leaves the contents of the target register <i>RT</i> undefined if the string length is 0.
mfsr	mfsr	This is a nonprivileged instruction in POWER® family. It is a privileged instruction in PowerPC®.
mfmsr	mfmsr	This is a nonprivileged instruction in POWER® family. It is a privileged instruction in PowerPC®.
mfdec	mfdec	The mfdec instruction is nonprivileged in POWER® family, but becomes a privileged instruction in PowerPC®. As a result, the DEC encoding number for the mfdec instruction is different for POWER® family and PowerPC®.
mffs	mffs	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.

See “Features of the AIX® assembler” on page 1 for more information on the PowerPC®-specific features of the assembler.

Related concepts:

“Migrating source programs” on page 143

The assembler issues errors and warnings if a source program contains instructions that are not in the current assembly mode.

“Functional differences for POWER® family and PowerPC® instructions”

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

“Differences between POWER® family and PowerPC® instructions with the same op code” on page 145

The differences between POWER® family and PowerPC® instructions with the same op code

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER® family and PowerPC®.

“POWER® family instructions deleted from PowerPC®” on page 151

The POWER® family instructions deleted from PowerPC®.

“Added PowerPC® instructions” on page 152

The instructions that have been added to PowerPC®, but are not in the POWER® family.

Differences between POWER® family and PowerPC® instructions with the same op code

The differences between POWER® family and PowerPC® instructions with the same op code

This section discusses the following:

Related concepts:

“Migrating source programs” on page 143

The assembler issues errors and warnings if a source program contains instructions that are not in the current assembly mode.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

“Differences between POWER® family and PowerPC® instructions with the same op code”

The differences between POWER® family and PowerPC® instructions with the same op code

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER® family and PowerPC®.

“POWER® family instructions deleted from PowerPC®” on page 151

The POWER® family instructions deleted from PowerPC®.

“Added PowerPC® instructions” on page 152

The instructions that have been added to PowerPC®, but are not in the POWER® family.

Instructions with the same op code, mnemonic, and function

The instructions are available in POWER® family and PowerPC®. These instructions share the same op code and mnemonic, and have the same function in a POWER® family and PowerPC®.

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**
- **cmpi**
- **cmpli**
- **cmpl**

The input operand format for POWER® family is:

BF, RA, SI | RB | UI

The input operand format for PowerPC® is:

BF, L, RA, SI | RB | 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 presets this bit to 0. For 32-bit PowerPC® platforms, this bit must be set to 0, or an invalid instruction form results.

Related concepts:

“Migrating source programs” on page 143

The assembler issues errors and warnings if a source program contains instructions that are not in the current assembly mode.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

“Differences between POWER® family and PowerPC® instructions with the same op code” on page 145

The differences between POWER® family and PowerPC® instructions with the same op code

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER® family and PowerPC®.

“POWER® family instructions deleted from PowerPC®” on page 151

The POWER® family instructions deleted from PowerPC®.

“Added PowerPC® instructions” on page 152

The instructions that have been added to PowerPC®, but are not in the POWER® family.

Instructions with the same op code and function

The instructions available in POWER® family and PowerPC® that share the same op code and function, but have different mnemonics and input operand formats.

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

POWER® family	PowerPC®
cal	addi
mtsri	mtsrin
svca	sc
cau	addis

Note:

1. **lil** is an extended mnemonic of **cal**, and **li** is an extended mnemonic of **addi**. Since the op code, function, and input operand format are the same, the assembler provides a cross-reference for **lil** and **li**.
2. **liu** is an extended mnemonic of **cau**, and **lis** is an extended mnemonic of **addis**. Since the input operand format is different, the assembler does not provide a cross-reference for **liu** and **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

The assembler processing of the **mfdec** instructions for each assembly mode.

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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Migrating source programs” on page 143

The assembler issues errors and warnings if a source program contains instructions that are not in the current assembly mode.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

“Differences between POWER® family and PowerPC® instructions with the same op code” on page 145

The differences between POWER® family and PowerPC® instructions with the same op code

“Extended mnemonics changes”

The extended mnemonics for POWER® family and PowerPC®.

“POWER® family instructions deleted from PowerPC®” on page 151

The POWER® family instructions deleted from PowerPC®.

“Added PowerPC® instructions” on page 152

The instructions that have been added to PowerPC®, but are not in the POWER® family.

Extended mnemonics changes

The extended mnemonics for POWER® family and PowerPC®.

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 “Extended instruction mnemonics” on page 118 for more information.

Related concepts:

“Migrating source programs” on page 143

The assembler issues errors and warnings if a source program contains instructions that are not in the current assembly mode.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

“Differences between POWER® family and PowerPC® instructions with the same op code” on page 145
The differences between POWER® family and PowerPC® instructions with the same op code

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER® family and PowerPC®.

“POWER® family instructions deleted from PowerPC®” on page 151

The POWER® family instructions deleted from PowerPC®.

“Added PowerPC® instructions” on page 152

The instructions that have been added to PowerPC®, but are not in the POWER® family.

Extended mnemonics in com mode

The extended mnemonics for branch, logical, load, and arithmetic instructions.

The following PowerPC® extended mnemonics for branch conditional instructions have been added:

- **bdzt**
- **bdzta**
- **bdztl**
- **bdztla**
- **bdzf**
- **bdzfa**
- **bdzfl**
- **bdzfla**
- **bdnzt**
- **bdnzta**
- **bdnztl**
- **bdnztla**
- **bdnzf**
- **bdnzfa**
- **bdnzfl**
- **bdnzfla**
- **bdztlr**
- **bdztlrl**
- **bdzflr**
- **bdzflrl**
- **bdnztlr**
- **bdnztlrl**
- **bdnzflr**
- **bdnzflrl**
- **bun**
- **buna**
- **bunl**
- **bunla**
- **bunlr**
- **bunlrl**
- **bunctr**
- **bunctrl**
- **bnu**
- **bnu**

- **bnul**
- **bnula**
- **bnulr**
- **bnulrl**
- **bnuctr**
- **bnuctrl**

The following PowerPC[®] extended mnemonics for condition register logical instructions have been added:

- **crset**
- **crclr**
- **crmove**
- **crnot**

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**
- **twlng**
- **twlngi**
- **twlnl**
- **twlnli**
- **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[.]**
- **rotlw[.]**
- **rotlwi[.]**
- **rotrwi[.]**
- **clrlwi[.]**
- **clrrwi[.]**
- **clrlslwi[.]**

Related concepts:

“Migrating source programs” on page 143

The assembler issues errors and warnings if a source program contains instructions that are not in the current assembly mode.

“Functional differences for POWER[®] family and PowerPC[®] instructions” on page 144

The POWER[®] family and PowerPC[®] instructions that share the same op code on POWER[®] family and PowerPC[®] platforms, but differ in their functional definition.

“Differences between POWER[®] family and PowerPC[®] instructions with the same op code” on page 145

The differences between POWER[®] family and PowerPC[®] instructions with the same op code

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER[®] family and PowerPC[®].

“POWER[®] family instructions deleted from PowerPC[®]” on page 151

The POWER[®] family instructions deleted from PowerPC[®].

“Added PowerPC[®] instructions” on page 152

The instructions that have been added to PowerPC[®], but are not in the POWER[®] family.

Extended mnemonics in ppc mode

The PowerPC[®] extended mnemonic for fixed-point arithmetic instruction.

The following PowerPC[®] extended mnemonic for fixed-point arithmetic instructions has been added for **ppc** mode:

- **sub**

Related concepts:

“Migrating source programs” on page 143

The assembler issues errors and warnings if a source program contains instructions that are not in the current assembly mode.

“Functional differences for POWER[®] family and PowerPC[®] instructions” on page 144

The POWER[®] family and PowerPC[®] instructions that share the same op code on POWER[®] family and PowerPC[®] platforms, but differ in their functional definition.

“Differences between POWER[®] family and PowerPC[®] instructions with the same op code” on page 145

The differences between POWER[®] family and PowerPC[®] instructions with the same op code

“POWER[®] family instructions deleted from PowerPC[®]” on page 151

The POWER[®] family instructions deleted from PowerPC[®].

“Added PowerPC[®] instructions” on page 152

The instructions that have been added to PowerPC[®], but are not in the POWER[®] family.

“Instructions available only for the PowerPC[®] 601 RISC microprocessor” on page 152

The PowerPC[®] instructions that are available only for the PowerPC[®] 601 RISC Microprocessor.

“Extended instruction mnemonics” on page 118

The assembler supports a set of extended mnemonics and symbols to simplify assembly language programming.

POWER[®] family instructions deleted from PowerPC[®]

The 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

Item	Description		
abs[o][.]	clcs	div[o][.]	divs[o][.]
doz[o][.]	dozi	lscbx[.]	maskg[.]
maskir[.]	mfmq	mfrtcl	mfrtcu
mtmq	mtrtcl	mtrtcu	mul[o][.]
nabs[o][.]	rlmi[.]	rrib[.]	sle[.]
sleq[.]	sliq[.]	slliq[.]	sllq[.]
slq[.]	sraiq[.]	sraq[.]	sre[.]
srea[.]	sreq[.]	sriq[.]	srliq[.]
srlq[.]	srq[.]	svcla	

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

Item	Description		
clf	cli	dclst	dclz
mfsdr0	mfsri	mftid	mtsdr0
mttid	rac[.]	rfsvc	svc
svcl	tlbi		

Related concepts:

“Migrating source programs” on page 143

The assembler issues errors and warnings if a source program contains instructions that are not in the current assembly mode.

“Functional differences for POWER[®] family and PowerPC[®] instructions” on page 144

The POWER[®] family and PowerPC[®] instructions that share the same op code on POWER[®] family and PowerPC[®] platforms, but differ in their functional definition.

“Differences between POWER[®] family and PowerPC[®] instructions with the same op code” on page 145

The differences between POWER[®] family and PowerPC[®] instructions with the same op code

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER[®] family and PowerPC[®].

“POWER[®] family instructions deleted from PowerPC[®]”

The POWER[®] family instructions deleted from PowerPC[®].

“Added PowerPC® instructions”

The instructions that have been added to PowerPC®, but are not in the POWER® family.

Added PowerPC® instructions

The instructions that have been added to PowerPC®, but are not in the POWER® family.

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

Item	Description		
dcbf	dcbi	dcbst	dcbt
dcbtst	dcbz	divw[o][.]	divwu[o][.]
eieio	extsb[.]	fadds[.]	fdivs[.]
fmadds[.]	fmsubs[.]	fmuls[.]	fnmadds[.]
fnmsubs[.]	fsubs[.]	icbi	lwarx
mfear	mfpvr	mfsprg	mfsrin
mtear	mtsprg	mulhw[.]	mulhwu[.]
stwcx.	subf[o][.]		

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

Item	Description		
mfdbatl	mfdbatu	mtdbatl	mtdbatu
mttb	mttbu	mftb	mftbu
mfibatl	mfibatu	mtibatl	mtibatu

Related concepts:

“Migrating source programs” on page 143

The assembler issues errors and warnings if a source program contains instructions that are not in the current assembly mode.

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER® family and PowerPC®.

“POWER® family instructions deleted from PowerPC®” on page 151

The POWER® family instructions deleted from PowerPC®.

“Instructions available only for the PowerPC® 601 RISC microprocessor”

The PowerPC® instructions that are available only for the PowerPC® 601 RISC Microprocessor.

Instructions available only for the PowerPC® 601 RISC microprocessor

The PowerPC® instructions that are 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 30. PowerPC® 601 RISC Microprocessor-Unique Instructions

Item	Description		
eciwx	ecowx	mfbatl	mfbatu
mtbatl	mtbatu	tlbie	

Note: Extended mnemonics, with the exception of **mfspr** and **mtspr** extended mnemonics, are not provided.

Related concepts:

“Migrating source programs” on page 143

The assembler issues errors and warnings if a source program contains instructions that are not in the current assembly mode.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

“Differences between POWER® family and PowerPC® instructions with the same op code” on page 145

The differences between POWER® family and PowerPC® instructions with the same op code

“Extended mnemonics changes” on page 147

The extended mnemonics for POWER® family and PowerPC®.

“POWER® family instructions deleted from PowerPC®” on page 151

The POWER® family instructions deleted from PowerPC®.

“Added PowerPC® instructions” on page 152

The instructions that have been added to PowerPC®, but are not in the POWER® family.

Migration of branch conditional statements with no separator after mnemonic

The AIX® assembler parses some statements different from the previous version of the assembler.

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.

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:

```

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):

```

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 concepts:

“Features of the AIX® assembler” on page 1

Features of AIX Assembler.

“Extended mnemonics of branch instructions” on page 119

The assembler supports extended mnemonics for different types of Register instructions.

Instruction set

This topic contains reference articles for the operating system assembler instruction set.

Updates to the Power Instruction Set Architecture (ISA) might have changed existing instructions, deprecated existing instructions, or added new instructions, as compared to the information contained in this document. See the latest version of the Power ISA documentation for updated information at <https://www.power.org/documentation>.

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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	///
21	OE
22 - 30	360
31	Rc

POWER® family

abs	<i>RT</i> , <i>RA</i>
abs.	<i>RT</i> , <i>RA</i>
abso	<i>RT</i> , <i>RA</i>
abso.	<i>RT</i> , <i>RA</i>

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
abs	0	None	0	None
abs.	0	None	1	LT,GT,EQ,SO
abso	1	SO,OV	0	None
abso.	1	SO,OV	1	LT,GT,EQ,SO

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

Item Description

<i>RT</i>	Specifies the target general-purpose register where result of operation is stored.
<i>RA</i>	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 0xB004 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

add (Add) or cax (Compute Address) instruction

Purpose

Adds the contents of two general-purpose registers.

Syntax

PowerPC®

```
add      RT, RA, RB
add.     RT, RA, RB
addo     RT, RA, RB
addo.    RT, RA, RB
```

POWER® family

```
cax      RT, RA, RB
cax.     RT, RA, RB
caxo     RT, RA, RB
caxo.    RT, RA, RB
```

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21	OE
22 - 30	266
31	Rc

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
add	0	None	0	None
add.	0	None	1	LT,GT,EQ,SO
addo	1	SO,OV	0	None
addo.	1	SO,OV	1	LT,GT,EQ,SO
cax	0	None	0	None
cax.	0	None	1	LT,GT,EQ,SO
caxo	1	SO,OV	0	None
caxo.	1	SO,OV	1	LT,GT,EQ,SO

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

Item Description

- 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

- 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.
```
- 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.
```
- 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.
```
- 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point address computation instructions” on page 24

The different address computation instructions in POWER® family are merged into the arithmetic instructions for PowerPC®.

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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21	OE
22 - 30	10
31	Rc

PowerPC®

addc *RT, RA, RB*

addc. *RT, RA, RB*

addco *RT, RA, RB*

addco. *RT, RA, RB*

Item **Description**

a *RT, RA, RB*

a. *RT, RA, RB*

ao *RT, RA, RB*

ao. *RT, RA, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
addc	0	CA	0	None
addc.	0	CA	1	LT,GT,EQ,SO
addco	1	SO,OV,CA	0	None
addco.	1	SO,OV,CA	1	LT,GT,EQ,SO
a	0	CA	0	None
a.	0	CA	1	LT,GT,EQ,SO
ao	1	SO,OV,CA	0	None
ao.	1	SO,OV,CA	1	LT,GT,EQ,SO

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

Item Description

- 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

- 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.
```
- 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.
```
- 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.
```
- 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21	OE
22 - 30	138
31	Rc

PowerPC®

adde *RT, RA, RB*
adde. *RT, RA, RB*
addeo *RT, RA, RB*
addeo. *RT, RA, RB*

POWER® family

ae *RT, RA, RB*
ae. *RT, RA, RB*
aeo *RT, RA, RB*
aeo. *RT, RA, RB*

Description

The **adde** 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 **adde** 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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
adde	0	CA	0	None
adde.	0	CA	1	LT,GT,EQ,SO
addeo	1	SO,OV,CA	0	None
addeo.	1	SO,OV,CA	1	LT,GT,EQ,SO
ae	0	CA	0	None
ae.	0	CA	1	LT,GT,EQ,SO
aeo	1	SO,OV,CA	0	None
aeo.	1	SO,OV,CA	1	LT,GT,EQ,SO

The four syntax forms of the **adde** 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

Item Description

- 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

- 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:


```
# 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.
```
- 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:


```
# 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.
```
- 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:


```
# 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.
```
- 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:

```
# 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	14
6 - 10	RT
11 - 15	RA
16 - 31	SI/D

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>D</i>	Specifies 16-bit two's complement integer sign extended to 32 bits.
<i>SI</i>	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:

```
# Assume GPR 5 contains 0x0000 0900.
cal 4,0xFFFF8FF0(5)
# GPR 4 now contains 0xFFFF 98F0.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point address computation instructions” on page 24

The different address computation instructions in POWER® family are merged into the arithmetic instructions for PowerPC®.

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 affects the Carry bit of the Fixed-Point Exception Register.

Syntax

Bits	Value
0 - 5	12
6 - 10	RT
11 - 15	RA
16 - 31	SI

PowerPC®

addic *RT, RA, SI*

POWER® family

ai *RT, RA, SI*

See Extended Mnemonics of Fixed-Point Arithmetic Instructions for more information.

Description

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>SI</i>	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:

```
# Assume GPR 4 contains 0x0000 2346.  
addic 6,4,0xFFFFFFFF  
# GPR 6 now contains 0x0000 2345.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	13
6 - 10	RT
11 - 15	RA
16 - 31	SI

PowerPC®

addic. *RT, RA, SI*

POWER® family

ai. *RT, RA, SI*

See Extended Mnemonics of Fixed-Point Arithmetic Instructions for more information.

Description

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>SI</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	15
6 - 10	<i>RT</i>
11 - 15	<i>RA</i>
16 - 31	<i>SI/UI</i>

Table 31. PowerPC

PowerPC	PowerPC
addis	<i>RT, RA, SI</i>
addis	<i>RT, D(RA)</i>

Table 32. POWER family

POWER family	POWER family
cau	<i>RT, RA, UI</i>

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 do not affect Condition Register Field 0 or the Fixed-Point Exception Register. The **cau** instruction has one syntax form. The **addis** instruction has two syntax forms; however, the second form is only valid when the R_TOCU relocation type is used in the *D* expression. The R_TOCU relocation type can be specified explicitly with the @u relocation specifier or implicitly by using a **QualName** parameter with a TE storage-mapping class.

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies first source general-purpose register for operation.
<i>UI</i>	Specifies 16-bit unsigned integer for operation.
<i>SI</i>	Specifies
16-bit signed integer for operation.	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:

```
# Assume GPR 6 contains 0x0000 4000.
addis 7,6,0x0011
# GPR 7 now contains 0x0011 4000.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point address computation instructions” on page 24

The different address computation instructions in POWER® family are merged into the arithmetic instructions for PowerPC®.

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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	///
21	OE
22 - 30	234
31	Rc

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
addme	0	CA	0	None
addme.	0	CA	1	LT,GT,EQ,SO
addmeo	1	SO,OV,CA	0	None
addmeo.	1	SO,OV,CA	1	LT,GT,EQ,SO
ame	0	CA	0	None
ame.	0	CA	1	LT,GT,EQ,SO
ameo	1	SO,OV,CA	0	None
ameo.	1	SO,OV,CA	1	LT,GT,EQ,SO

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

Item Description

- RT* Specifies target general-purpose register where result of operation is stored.
RA Specifies source general-purpose register for operation.

Examples

- 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.
```
- 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.
```
- 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.
addmeo 6,4
# GPR 6 now contains 0x7FFF FFFF.
```
- 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.
addmeo. 6,4
# GPR 6 now contains 0x8000 000.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	///
21	OE
22 - 30	202
31	Rc

PowerPC®

addze *RT, RA*
addze. *RT, RA*
addzeo *RT, RA*
addzeo. *RT, RA*

POWER® family

aze *RT, RA*
aze. *RT, RA*
azeo *RT, RA*
azeo. *RT, RA*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
addze	0	CA	0	None
addze.	0	CA	1	LT,GT,EQ,SO
addzeo	1	SO,OV,CA	0	None
addzeo.	1	SO,OV,CA	1	LT,GT,EQ,SO
aze	0	CA	0	None
aze.	0	CA	1	LT,GT,EQ,SO
azeo	1	SO,OV,CA	0	None
azeo.	1	SO,OV,CA	1	LT,GT,EQ,SO

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

Item Description

- RT* Specifies target general-purpose register where result of operation is stored.
RA Specifies source general-purpose register for operation.

Examples

- 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.
addze 6,4
# GPR 6 now contains 0x7B41 92C0.
```
- 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.
addze. 6,4
# GPR 6 now contains 0xF000 0000.
```
- 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.
```
- 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

and (AND) instruction

Purpose

Logically ANDs the contents of two general-purpose registers and places the result in a general-purpose register.

Syntax

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	28
31	Rc

Item	Description
and	<i>RA, RS, RB</i>
and.	<i>RA, RS, RB</i>

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
and	None	None	0	None
and.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **and** 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

Item	Description
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

- The following code logically ANDs the contents of GPR 4 with the contents of GPR 7 and stores the result in GPR 6:

```
# Assume GPR 4 contains 0xFFF2 5730.
# Assume GPR 7 contains 0x7B41 92C0.
and 6,4,7
# GPR 6 now contains 0x7B40 1200.
```
- 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:

```
# Assume GPR 4 contains 0xFFF2 5730.
# Assume GPR 7 contains 0xFFFF EFFF.
and. 6,4,7
# GPR 6 now contains 0xFFF2 4730.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	60
31	Rc

Item	Description
andc	<i>RA, RS, RB</i>
andc.	<i>RA, RS, RB</i>

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
andc	None	None	0	None
andc.	None	None	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

andi. or andil. (AND Immediate) instruction

Purpose

Logically ANDs the contents of a general-purpose register with an immediate value.

Syntax

Bits	Value
0 - 5	28
6 - 10	RS
11 - 15	RA
16 - 31	UI

PowerPC®

andi. *RA, RS, UI*

POWER® family

andil. *RA, RS, UI*

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>UI</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

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

Bits	Value
0 - 5	29
6 - 10	RS
11 - 15	RA
16 - 31	UI

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

Item	Description

<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>UI</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

b (Branch) instruction

Purpose

Branches to a specified target address.

Syntax

Bits	Value
0 - 5	18
6 - 29	LL
30	AA
31	LK

Item	Description
b	<i>target_address</i>
ba	<i>target_address</i>
bl	<i>target_address</i>
bla	<i>target_address</i>

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.

Item	Description			
Syntax Form	Absolute Address Bit (AA)	Fixed-Point Exception Register	Link Bit (LK)	Condition Register Field 0
b	0	None	0	None
ba	1	None	0	None
bl	0	None	1	None
bla	1	None	1	None

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

Item	Description
<i>target_address</i>	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 concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“b (Branch) instruction” on page 176

bc (Branch Conditional) instruction

Purpose

Conditionally branches to a specified target address.

Syntax

Bits	Value
0 - 5	16
6 - 10	BO
11 - 15	BI
16 - 29	BD
30	AA
31	LK

Item	Description
bc	<i>BO, BI, target_address</i>
bca	<i>BO, BI, target_address</i>
bcl	<i>BO, BI, target_address</i>
bcla	<i>BO, BI, target_address</i>

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.

Item	Description			
Syntax Form	Absolute Address Bit (AA)	Fixed-Point Exception Register	Link Bit (LK)	Condition Register Field 0
bc	0	None	0	None
bca	1	None	0	None
bcl	0	None	1	None
bcla	1	None	1	None

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 33. BO Field Values Using pre-V2.00 Encoding

BO	Description
0000y	Decrement the CTR; then branch if the decremented CTR is not 0 and the condition is False.
0001y	Decrement the CTR; then branch if the decremented CTR is 0 and the condition is False.
001zy	Branch if the condition is False.
0100y	Decrement the CTR; then branch if bits the decremented CTR is not 0 and the condition is True.
0101y	Decrement the CTR; then branch if the decremented CTR is 0 and the condition is True.
011zy	Branch if the condition is True.
1z00y	Decrement the CTR; then branch if the decremented CTR is not 0.
1z01y	Decrement the CTR; then branch if the decremented CTR is 0.
1z1zz	Branch always.

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

BO	Description
0000z	Decrement the CTR; then branch if the decremented CTR is not 0 and the condition is False.
0001z	Decrement the CTR; then branch if the decremented CTR is 0 and the condition is False.
001at	Branch if the condition is False.
0100z	Decrement the CTR; then branch if bits the decremented CTR is not 0 and the condition is True.
0101z	Decrement the CTR; then branch if the decremented CTR is 0 and the condition is True.
011at	Branch if the condition is True.
1a00t	Decrement the CTR; then branch if the decremented CTR is not 0.
1a01t	Decrement the CTR; then branch if the decremented CTR is 0.
1z1zz	Branch always.

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:

Item	Description
at	Hint
00	No hint is given.
01	Reserved
10	The branch is likely not to be taken.
11	The branch is likely to be taken.

Parameters

Item	Description
<i>target_address</i>	Specifies the target address. For absolute branches such as bca and bcla , the target address can be immediate data containable in 16 bits.
<i>BI</i>	Specifies bit in Condition Register for condition comparison.
<i>BO</i>	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 concepts:

“Assembler overview” on page 1

The assembler program takes machine-language instructions and translates them into machine object-code.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“b (Branch) instruction” on page 176

bcctr or bcc (Branch Conditional to Count Register) instruction

Purpose

Conditionally branches to the address contained within the Count Register.

Syntax

Bits	Value
0 - 5	19
6 - 10	BO
11 - 15	BI
16 - 18	///
19 - 20	BH
21 - 30	528
31	LK

PowerPC®

bcctr *BO, BI, BH*

bcctrl *BO, BI, BH*

POWER® family

bcc *BO, BI, BH*

bccl *BO, BI, BH*

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.

Item	Description			
Syntax Form	Absolute Address Bit (AA)	Fixed-Point Exception Register	Link Bit (LK)	Condition Register Field 0
bcctr	None	None	0	None
bcctrl	None	None	1	None
bcc	None	None	0	None
bccl	None	None	1	None

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:

BO	Description
0000y	Decrement the CTR; then branch if the decremented CTR is not 0 and the condition is False.
0001y	Decrement the CTR; then branch if the decremented CTR is 0 and the condition is False.
001zy	Branch if the condition is False.
0100y	Decrement the CTR; then branch if bits the decremented CTR is not 0 and the condition is True.
0101y	Decrement the CTR; then branch if the decremented CTR is 0 and the condition is True.
011zy	Branch if the condition is True.
1z00y	Decrement the CTR; then branch if the decremented CTR is not 0.
1z01y	Decrement the CTR; then branch if the decremented CTR is 0.
1z1zz	Branch always.

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 35. BO Field Values Using V2.00 Encoding

BO	Description
0000z	Decrement the CTR; then branch if the decremented CTR is not 0 and the condition is False.
0001z	Decrement the CTR; then branch if the decremented CTR is 0 and the condition is False.
001at	Branch if the condition is False.
0100z	Decrement the CTR; then branch if bits the decremented CTR is not 0 and the condition is True.
0101z	Decrement the CTR; then branch if the decremented CTR is 0 and the condition is True.
011at	Branch if the condition is True.
1a00t	Decrement the CTR; then branch if the decremented CTR is not 0.
1a01t	Decrement the CTR; then branch if the decremented CTR is 0.
1z1zz	Branch always.

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:

at	Hint
00	No hint is given.
01	Reserved
10	The branch is very likely not to be taken.
11	The branch is very likely to be taken.

For more information, see the IBM Power ISA PDF.

The Branch Hint field (BH) is used to provide a hint about the use of the instruction, as shown below:

BH	Hint
00	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.
01	Reserved
10	Reserved
11	The target address is not predictable.

Parameters

Item	Description
<i>BO</i>	Specifies Branch Option field.
<i>BI</i>	Specifies bit in Condition Register for condition comparison.
<i>BIF</i>	Specifies the Condition Register field that specifies the Condition Register bit (LT, GT, EQ, or SO) to be used for condition comparison.
<i>BH</i>	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:

```
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.
bcctrl 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 concepts:

“Assembler overview” on page 1

The assembler program takes machine-language instructions and translates them into machine object-code.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“b (Branch) instruction” on page 176

“Extended mnemonics for branch prediction” on page 124

The assembler source program can have information on branch conditional instruction by adding a branch prediction suffix to the mnemonic of the instruction.

bclr or bcr (Branch Conditional Link Register) instruction

Purpose

Conditionally branches to an address contained in the Link Register.

Syntax

Bits	Value
0 - 5	19
6 - 10	BO
11 - 15	BI
16 - 18	///
19 - 20	BH
21 - 30	16
31	LK

PowerPC®

bclr *BO, BI, BH*
bclrl *BO, BI, BH*

POWER® family

bcr *BO, BI, BH*
bclrl *BO, BI, BH*

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.

Syntax Form	Absolute Address Bit (AA)	Fixed-Point Exception Register	Link Bit (LK)	Condition Register Field 0
bclr	None	None	0	None
bclrl	None	None	1	None
bcr	None	None	0	None
bclrl	None	None	1	None

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:

BO	Description
0000y	Decrement the CTR; then branch if the decremented CTR is not 0 and the condition is False.
0001y	Decrement the CTR; then branch if the decremented CTR is 0 and the condition is False.
001zy	Branch if the condition is False.
0100y	Decrement the CTR; then branch if the decremented CTR is not 0 and the condition is True.
0101y	Decrement the CTR; then branch if the decremented CTR is 0 and the condition is True.
011zy	Branch if the condition is True.
1z00y	Decrement the CTR; then branch if the decremented CTR is not 0.
1z01y	Decrement the CTR; then branch if the decremented CTR is 0.
1z1zz	Branch always.

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 36. BO Field Values Using V2.00 Encoding

BO	Description
0000z	Decrement the CTR; then branch if the decremented CTR is not 0 and the condition is False.
0001z	Decrement the CTR; then branch if the decremented CTR is 0 and the condition is False.
001at	Branch if the condition is False.
0100z	Decrement the CTR; then branch if bits the decremented CTR is not 0 and the condition is True.
0101z	Decrement the CTR; then branch if the decremented CTR is 0 and the condition is True.
011at	Branch if the condition is True.
1a00t	Decrement the CTR; then branch if the decremented CTR is not 0.
1a01t	Decrement the CTR; then branch if the decremented CTR is 0.
1z1zz	Branch always.

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:

at	Hint
00	No hint is given.
01	Reserved
01	The branch is very likely not to be taken.
11	The branch is very likely to be taken.

The Branch Hint field (BH) is used to provide a hint about the use of the instruction, as shown below:

BH	Hint
00	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.
01	Reserved
10	Reserved
11	The target address is not predictable.

Parameters

Item	Description
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:

```
bclr 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'.
```

clcs (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

Bits	Value
0-5	31
6-10	RT
11-15	RA
16-20	///
21-30	531
31	Rc

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*.

Item	Description
Value of RA	Cache Line Size Returned in RT
00xxx	Undefined
010xx	Undefined
01100	Instruction Cache Line Size
01101	Data Cache Line Size
01110	Minimum Cache Line Size
01111	Maximum Cache Line Size
1xxxx	Undefined

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

Item Description

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:

```
# Assume that 0xf is the cache  
line size requested  
.  
    clcs 4,0xf  
# GPR 4 now contains the maximum Cache Line size.
```

Related concepts:

“clf (Cache Line Flush) instruction” on page 186

“dcbf (Data Cache Block Flush) instruction” on page 205

“dcbi (Data Cache Block Invalidate) instruction” on page 206

“dcbtst (Data Cache Block Touch for Store) instruction” on page 212
 “dcbz or dclz (Data Cache Block Set to Zero) instruction” on page 214
 “dclst (Data Cache Line Store) instruction” on page 216
 “icbi (Instruction Cache Block Invalidate) instruction” on page 282
 “isync or ics (Instruction Synchronize) instruction” on page 283
 “Processing and storage” on page 9
 The processor stores the data in the main memory and in the registers.

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

Bits	Value
0-5	31
6-10	///
11-15	RA
16-20	RB
21-30	118
31	Rc

POWER® family

clf *RA, RB*

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

Item	Description
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:

```
# 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
clf     R4,R3          # Flush A from cache to main memory
dcs     # Ensure clf 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 concepts:

“clcs (Cache Line Compute Size) instruction” on page 184

“cli (Cache Line Invalidate) instruction”

“dcbf (Data Cache Block Flush) instruction” on page 205

“dcbt (Data Cache Block Touch) instruction” on page 209

“dcbst (Data Cache Block Store) instruction” on page 208

“dcbz or dclz (Data Cache Block Set to Zero) instruction” on page 214

“dclst (Data Cache Line Store) instruction” on page 216

“icbi (Instruction Cache Block Invalidate) instruction” on page 282

“sync (Synchronize) or dcs (Data Cache Synchronize) instruction” on page 498

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

cli (Cache Line Invalidate) instruction

Purpose

Invalidates 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

Bits	Value
0-5	31
6-10	///
11-15	RA
16-20	RB
21-30	502
31	Rc

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

Item	Description
<i>RA</i>	Specifies the source general-purpose register for EA calculation and possibly the target general-purpose register (when <i>RA</i> is not GPR 0) for operation.
<i>RB</i>	Specifies the source general-purpose register for EA calculation.

Security

The **cli** instruction is privileged.

Related concepts:

“clcs (Cache Line Compute Size) instruction” on page 184

“clf (Cache Line Flush) instruction” on page 186

“cli (Cache Line Invalidate) instruction” on page 187

“dcbt (Data Cache Block Touch) instruction” on page 209

“dcbtst (Data Cache Block Touch for Store) instruction” on page 212

“dcbz or dclz (Data Cache Block Set to Zero) instruction” on page 214

“dclst (Data Cache Line Store) instruction” on page 216

“icbi (Instruction Cache Block Invalidate) instruction” on page 282

“sync (Synchronize) or dcs (Data Cache Synchronize) instruction” on page 498

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“dcbf (Data Cache Block Flush) instruction” on page 205

cmp (Compare) instruction

Purpose

Compares the contents of two general-purpose registers algebraically.

Syntax

Bits	Value
0-5	31
6-8	BF
9	/
10	L
11-15	RA
16-20	RB
21-30	0
31	/

Item	Description
cmp	<i>BF, L, RA, RB</i>

See Extended Mnemonics of Fixed-Point Compare Instructions for more information.

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:

Item	Description	
Bit	Name	Description
0	LT	(RA) < SI
1	GT	(RA) > SI
2	EQ	(RA) = SI
3	SO	SO,OV

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

Item	Description
<i>BF</i>	Specifies Condition Register Field 0-7 which indicates result of compare.
<i>L</i>	Must be set to 0 for the 32-bit subset architecture.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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:

```
# 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 concepts:

“cmpi (Compare Immediate) instruction” on page 191

“cmpli (Compare Logical Immediate) instruction” on page 193

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

cmpi (Compare Immediate) instruction

Purpose

Compares the contents of a general-purpose register and a given value algebraically.

Syntax

Bits	Value
0-5	11
6-8	BF
9	/
10	L
11-15	RA
16-31	SI

Item	Description
cmpi	<i>BF, L, RA, SI</i>

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:

Item	Description	
Bit	Name	Description
0	LT	(RA) < SI
1	GT	(RA) > SI
2	EQ	(RA) = SI
3	SO	SO,OV

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

Item	Description
<i>BF</i>	Specifies Condition Register Field 0-7 which indicates result of compare.
<i>L</i>	Must be set to 0 for the 32-bit subset architecture.
<i>RA</i>	Specifies first source general-purpose register for operation.
<i>SI</i>	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 concepts:

“cmp (Compare) instruction” on page 189

“cmpl (Compare Logical) instruction”

“cmpli (Compare Logical Immediate) instruction” on page 193

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

cmpl (Compare Logical) instruction

Purpose

Compares the contents of two general-purpose registers logically.

Syntax

Bits	Value
0-5	31
6-8	BF
9	/
10	L
11-15	RA
16-20	RB
21-30	32
31	/

Item	Description
cmpl	<i>BF, L, RA, RB</i>

See Extended Mnemonics of Fixed-Point Compare Instructions for more information.

Description

The **cmpl** instruction compares the contents of general-purpose register (GPR) *RA* with the contents of GPR *RB* as unsigned 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:

Item	Description	
Bit	Name	Description
0	LT	(RA) < SI
1	GT	(RA) > SI
2	EQ	(RA) = SI
3	SO	SO,OV

The **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 *BF* by the programmer.

Parameters

Item	Description
<i>BF</i>	Specifies Condition Register Field 0-7 which indicates result of compare.
<i>L</i>	Must be set to 0 for the 32-bit subset architecture.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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:

```
# 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 concepts:

“cmp (Compare) instruction” on page 189

“cmpi (Compare Immediate) instruction” on page 191

“cmpli (Compare Logical Immediate) instruction”

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

cmpli (Compare Logical Immediate) instruction

Purpose

Compares the contents of a general-purpose register and a given value logically.

Syntax

Bits	Value
0-5	10
6-8	BF
9	/
10	L
11-15	RA
16-31	UI

Item	Description
cmpli	<i>BF, L, RA, UI</i>

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:

Item	Description	
Bit	Name	Description
0	LT	(RA) < SI
1	GT	(RA) > SI
2	EQ	(RA) = SI
3	SO	SO,OV

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

Item	Description
<i>BF</i>	Specifies Condition Register Field 0-7 that indicates result of compare.
<i>L</i>	Must be set to 0 for the 32-bit subset architecture.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>UI</i>	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 concepts:

“cmp (Compare) instruction” on page 189

“cmpi (Compare Immediate) instruction” on page 191

“cmpli (Compare Logical Immediate) instruction” on page 193

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

cntlzd (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

Bits	Value
0-5	31
6-10	S
11-15	A
16-20	00000
21-30	58
31	Rc

PowerPC64

cntlzd *rA*, *rS* (Rc=0)

cntlzd. *rA*, *rS*(Rc=1)

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

Item	Description
<i>RA</i>	Specifies the target general purpose register for the results of the instruction.
<i>RS</i>	Specifies the source general purpose register containing the doubleword 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

Counts the number of leading zeros of the 32-bit value in a source general-purpose register (GPR) and stores the result in a GPR.

Syntax

Bits	Value
0 - 5	31
6-10	<i>RS</i>
11-15	<i>RA</i>
16-20	///
21-30	26
31	<i>Rc</i>

PowerPC®

cntlzw	<i>RA, RS</i>
cntlzw.	<i>RA, RS</i>

POWER® family

cntlz	<i>RA, RS</i>
cntlz.	<i>RA, RS</i>

Description

The **cntlzw** and **cntlz** instructions count the number (0 - 32) of consecutive zero bits of the 32 low-order bits of GPR *RS* and store the result in the target GPR *RA*.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
cntlzw	None	None	0	None
cntlzw.	None	None	1	LT,GT,EQ,SO
cntlz	None	None	0	None
cntlz.	None	None	1	LT,GT,EQ,SO

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

Item	Description

<i>RA</i>	Specifies the target general-purpose register where the result of the operation is stored.

<i>RS</i>	Specifies the source general-purpose register for the operation.

Examples

The following code counts the number of leading zeros in the 32-bit value contained in GPR 3 and places the result back in GPR 3:

```
# Assume GPR 3 contains 0xFFFF FFFF 0061 9920.
```

```
cntlzw 3,3
```

```
# GPR 3 now holds 0x0000 0000 0000 0009. Note that the high-order 32 bits  
are ignored when computing the result.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

crand (Condition Register AND) instruction

Purpose

Places the result of ANDing two Condition Register bits in a Condition Register bit.

Syntax

Bits	Value
0-5	19
6-10	BT
11-15	BA
16-20	BB
21-30	257
31	/

Item	Description

crand	<i>BT, BA, BB</i>

Description

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

Item	Description
<i>BT</i>	Specifies target Condition Register bit where result of operation is stored.
<i>BA</i>	Specifies source Condition Register bit for operation.
<i>BB</i>	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:

```
# 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 concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Condition register instructions” on page 21

The condition register instructions copy one CR field to another CR field or perform logical operations on CR bits.

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

Bits	Value
0-5	19
6-10	BT
11-15	BA
16-20	BB
21-30	129
31	/

Item	Description
crandc	<i>BT, BA, BB</i>

Description

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

Item	Description

<i>BT</i>	Specifies target Condition Register bit where result of operation is stored.

<i>BA</i>	Specifies source Condition Register bit for operation.

<i>BB</i>	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:

```
# 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 concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Condition register instructions” on page 21

The condition register instructions copy one CR field to another CR field or perform logical operations on CR bits.

creqv (Condition Register Equivalent) instruction

Purpose

Places the complemented result of XORing two Condition Register bits in a Condition Register bit.

Syntax

Bits	Value
0-5	19
6-10	BT
11-15	BA
16-20	BB
21-30	289
31	/

Item	Description

creqv	<i>BT, BA, BB</i>

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

Item	Description
<i>BT</i>	Specifies target Condition Register bit where result of operation is stored.
<i>BA</i>	Specifies source Condition Register bit for operation.
<i>BB</i>	Specifies source Condition Register bit for operation.

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 concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Condition register instructions” on page 21

The condition register instructions copy one CR field to another CR field or perform logical operations on CR bits.

crnand (Condition Register NAND) instruction

Purpose

Places the complemented result of ANDing two Condition Register bits in a Condition Register bit.

Syntax

Bits	Value
0-5	19
6-10	BT
11-15	BA
16-20	BB
21-30	225
31	/

Item	Description
crnand	<i>BT, BA, BB</i>

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

Item	Description

<i>BT</i>	Specifies target Condition Register bit where result of operation is stored.

<i>BA</i>	Specifies source Condition Register bit for operation.

<i>BB</i>	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:

```
# 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 concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Condition register instructions” on page 21

The condition register instructions copy one CR field to another CR field or perform logical operations on CR bits.

crnor (Condition Register NOR) instruction

Purpose

Places the complemented result of ORing two Condition Register bits in a Condition Register bit.

Syntax

Bits	Value
0-5	19
6-10	BT
11-15	BA
16-20	BB
21-30	33
31	/

Item	Description

crnor	<i>BT, BA, BB</i>

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

Item	Description
<i>BT</i>	Specifies target Condition Register bit where result of operation is stored.
<i>BA</i>	Specifies source Condition Register bit for operation.
<i>BB</i>	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.
crnor 4,8,4
# Condition Register bit 4 is now 0.
```

Related concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Condition register instructions” on page 21

The condition register instructions copy one CR field to another CR field or perform logical operations on CR bits.

cror (Condition Register OR) instruction

Purpose

Places the result of ORing two Condition Register bits in a Condition Register bit.

Syntax

Bits	Value
0-5	19
6-10	BT
11-15	BA
16-20	BB
21-30	449
31	/

Item	Description
cror	<i>BT, BA, BB</i>

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

Item	Description

<i>BT</i>	Specifies target Condition Register bit where result of operation is stored.

<i>BA</i>	Specifies source Condition Register bit for operation.

<i>BB</i>	Specifies 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.
```

```
crcr 4,8,4
```

```
# Condition Register bit 4 is now 1.
```

Related concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Condition register instructions” on page 21

The condition register instructions copy one CR field to another CR field or perform logical operations on CR bits.

crcrc (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

Bits	Value
0-5	19
6-10	BT
11-15	BA
16-20	BB
21-30	417
31	/

Item	Description

crcrc	<i>BT, BA, BB</i>

Description

The **crcrc** 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 **crcrc** instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters

Item	Description
<i>BT</i>	Specifies target Condition Register bit where result of operation is stored.
<i>BA</i>	Specifies source Condition Register bit for operation.
<i>BB</i>	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:

```
# Assume Condition Register bit 8 is 1.
# Assume Condition Register bit 4 is 0.
ciorc 4,8,4
# Condition Register bit 4 is now 1.
```

Related concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Condition register instructions” on page 21

The condition register instructions copy one CR field to another CR field or perform logical operations on CR bits.

crxor (Condition Register XOR) instruction

Purpose

Places the result of XORing two Condition Register bits in a Condition Register bit.

Syntax

Bits	Value
0-5	19
6-10	BT
11-15	BA
16-20	BB
21-30	193
31	/

Item	Description
crxor	<i>BT, BA, BB</i>

See Extended Mnemonics of Condition Register Logical Instructions for more information.

Description

The **crxor** instruction logically XORs 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 **crxor** instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters

Item	Description

<i>BT</i>	Specifies target Condition Register bit where result of operation is stored.

<i>BA</i>	Specifies source Condition Register bit for operation.

<i>BB</i>	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:

```
# Assume Condition Register bit 8 is 1.  
# Assume Condition Register bit 4 is 1.  
crxor 4,8,4  
# Condition Register bit 4 is now 0.
```

Related concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Condition register instructions” on page 21

The condition register instructions copy one CR field to another CR field or perform logical operations on CR bits.

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

Bits	Value
0-5	31
6-10	///
11-15	RA
16-20	RB
21-30	86
31	/

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

Item	Description
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 concepts:

- “clcs (Cache Line Compute Size) instruction” on page 184
- “clf (Cache Line Flush) instruction” on page 186
- “dcbt (Data Cache Block Touch) instruction” on page 209
- “dcbtst (Data Cache Block Touch for Store) instruction” on page 212
- “dcbz or dclz (Data Cache Block Set to Zero) instruction” on page 214
- “dclst (Data Cache Line Store) instruction” on page 216
- “icbi (Instruction Cache Block Invalidate) instruction” on page 282
- “sync (Synchronize) or dcs (Data Cache Synchronize) instruction” on page 498

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

Bits	Value
0-5	31
6-10	///
11-15	RA
16-20	RB
21-30	470
31	/

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

Item	Description

<i>RA</i>	Specifies the source general-purpose register for EA computation.

<i>RB</i>	Specifies the source general-purpose register for EA computation.

Security

The **dcbi** instruction is privileged.

Related concepts:

“clcs (Cache Line Compute Size) instruction” on page 184

“clf (Cache Line Flush) instruction” on page 186

“cli (Cache Line Invalidate) instruction” on page 187

“dcbi (Data Cache Block Invalidate) instruction” on page 206

“dcbtst (Data Cache Block Touch for Store) instruction” on page 212

“dcbz or dclz (Data Cache Block Set to Zero) instruction” on page 214

“dclst (Data Cache Line Store) instruction” on page 216

“icbi (Instruction Cache Block Invalidate) instruction” on page 282

“sync (Synchronize) or dcs (Data Cache Synchronize) instruction” on page 498

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

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

Bits	Value
0-5	31
6-10	///
11-15	RA
16-20	RB
21-30	54
31	/

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

Item	Description
<i>RA</i>	Specifies the source general-purpose register for EA computation.
<i>RB</i>	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.

```

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 for POWER5™ and later architecture.

Syntax

Bits	Value
0-5	31
6-10	TH
11-15	RA
16-20	RB
21-30	278
31	/

POWER5™

dcbt *RA , RB, TH*

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 (*TH*) field 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 depend on the target architecture selected with the **-m** flag or the .machine pseudo-op. The encodings of the *TH* field on POWER5™ and subsequent architectures are as follows:

TH Values	Description
0000	The program will probably soon load from the byte addressed by EA.
0001	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. The sequentially preceding blocks are the bytes addressed by EA + n * block_size, where n = 0, 1, 2, and so on.
0011	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. The sequentially preceding blocks are the bytes addressed by EA - n * block_size where n = 0, 1, 2, and so on.
1000	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 37.
1010	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 38 on page 211.

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 37. EA Encoding when TH=0b1000

Bit(s)	Name	Description
0-56	EA_TRUNC	High-order 57 bits of the effective address of the first unit of the data stream.
57	D	Direction 0 Subsequent units are the sequentially following units. 1 Subsequent units are the sequentially preceding units.
58	UG	0 No information is provided by the UG field. 1 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.
59	Reserved	Reserved
60-63	ID	Stream ID to use for this stream.

Table 38. EA Encoding when TH=0b1010

Bit(s)	Name	Description
0-31	Reserved	Reserved
32	GO	0 No information is provided by the GO field 1 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.
33-34	S	Stop 00 No information is provided by the S field. 01 Reserved 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). 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).
35-46	Reserved	Reserved
47-56	UNIT_CNT	Number of units in the data stream.
57	T	0 No information is provided by the T field. 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).
58	U	0 No information is provided by the U field. 1 The number of units in the data stream is unlimited (and the UNIT_CNT field is ignored).
59	Reserved	Reserved
60-63	ID	Stream ID to use for this stream.

The **dcbt** instruction has one syntax form and does not affect the Condition Register field 0 or the Fixed-Point Exception register.

Parameters

Item	Description
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:

```
# 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
                        # cache block.
    addi    5,4,32       # Compute address of second
                        # 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
                           # cache block.
        lwz     3,4,0        # Set sum = element number 1.
bigloop:
        addi    8,8,-1       # Decrement outer loop count
                           # and set CR field 0.
        mtspr   CTR,7        # Set counter (CTR) for
                           # inner loop.
        addi    5,5,32       # Computer address for next
                           # touch.
lttlloop:
        lwzu    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 concepts:

“clcs (Cache Line Compute Size) instruction” on page 184

“clf (Cache Line Flush) instruction” on page 186

“cli (Cache Line Invalidate) instruction” on page 187

“dcbtst (Data Cache Block Touch for Store) instruction”

“dcbz or dclz (Data Cache Block Set to Zero) instruction” on page 214

“dclst (Data Cache Line Store) instruction” on page 216

“icbi (Instruction Cache Block Invalidate) instruction” on page 282

“sync (Synchronize) or dcs (Data Cache Synchronize) instruction” on page 498

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

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.

Syntax

Bits	Value
0-5	31
6-10	TH
11-15	RA
16-20	RB
21-30	246
31	/

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 **dcbt** (Data Cache Block Touch) instruction.

The **dcbtst** instruction has one syntax form and does not affect Condition Register field 0 or the Fixed-Point Exception register.

The Touch Hint (*TH*) field is used to provide a hint that the program will probably store soon to 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 depend on the target architecture selected with the **-m** flag or the **.machine** pseudo-op. The encodings of the *TH* field are the same as for the **dcbt** instruction.

The **dcbtst** instruction serves as both a basic and extended mnemonic. The **dcbtst** mnemonic with three operands is the basic form, and the **dcbtst** with two operands is the extended form. In the extended form, the *TH* operand is omitted and assumed to be 0. The encodings of the *TH* field on POWER5™ and subsequent architectures are as follows:

TH Values	Description
0000	The program will probably store to the byte addressed by EA.
0001	The program will probably store to the data stream consisting of the block containing the byte addressed by EA and an unlimited number of sequentially following blocks. The sequentially preceding blocks are the bytes addressed by EA + n * block_size, where n = 0, 1, 2, and so on.
0011	The program will probably store to the data stream consisting of the block containing the byte addressed by EA and an unlimited number of sequentially preceding blocks. The sequentially preceding blocks are the bytes addressed by EA - n * block_size where n = 0, 1, 2, and so on.
1000	The dcbt instruction provides a hint that describes certain attributes of a data stream, and optionally indicates that the program will probably store to the stream. The EA is interpreted as described in EA Encoding when TH=0b1000 .
1010	The dcbt instruction provides a hint that describes certain attributes of a data stream, or indicates that the program will probably store to data streams that have been described using dcbt instructions in which TH[0] = 1 or probably no longer store to such data streams. The EA is interpreted as described in EA Encoding when TH=0b1010

Parameters

Item	Description
<i>RA</i>	Specifies source general-purpose register for operation.
<i>RB</i>	Specifies source general-purpose register for operation.
<i>TH</i>	Indicates when a sequence of data cache blocks might be modified.

Related concepts:

“clcs (Cache Line Compute Size) instruction” on page 184

“clf (Cache Line Flush) instruction” on page 186

“cli (Cache Line Invalidate) instruction” on page 187

“dcbst (Data Cache Block Store) instruction” on page 208

“dcbt (Data Cache Block Touch) instruction” on page 209

“dcbz or dclz (Data Cache Block Set to Zero) instruction”

“dclst (Data Cache Line Store) instruction” on page 216

“icbi (Instruction Cache Block Invalidate) instruction” on page 282

“sync (Synchronize) or dcs (Data Cache Synchronize) instruction” on page 498

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“isync or ics (Instruction Synchronize) instruction” on page 283

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

Purpose

The PowerPC® instruction, **dcbz**, sets all bytes of a cache block to 0.

The POWER® family instruction, **dclz**, sets all bytes of a cache line to 0.

Syntax

Bits	Value
0-5	31
6-10	///
11-15	RA
16-20	RB
21-30	1014
31	/

PowerPC®

dcbz *RA, RB*

POWER® family
dclz *RA, RB*

Description

The **dcbz** and **dclz** instructions work with data cache blocks and data cache lines respectively. If *RA* is not 0, the **dcbz** and **dclz** instructions compute an effective address (EA) by adding the contents of general-purpose register (GPR) *RA* to the contents of GPR *RB*. If GPR *RA* is 0, the EA is the contents of GPR *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 **dclz**, if GPR *RA* is not 0, the EA replaces the content of GPR *RA*.

The **dcbz** and **dclz** instructions are treated as a store to the addressed cache block or line with respect to protection.

The **dcbz** and **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

PowerPC®

<i>RA</i>	Specifies the source register for EA computation.
<i>RB</i>	Specifies the source register for EA computation.

POWER® family

<i>RA</i>	Specifies the source register for EA computation and the target register for EA update.
<i>RB</i>	Specifies the source register for EA computation.

Security

The **dclz** instruction is privileged.

Related concepts:

“clcs (Cache Line Compute Size) instruction” on page 184

“clf (Cache Line Flush) instruction” on page 186

“dcbi (Data Cache Block Invalidate) instruction” on page 206

“dcbst (Data Cache Block Store) instruction” on page 208

“dcbt (Data Cache Block Touch) instruction” on page 209

“dcbtst (Data Cache Block Touch for Store) instruction” on page 212

“dclst (Data Cache Line Store) instruction” on page 216

“icbi (Instruction Cache Block Invalidate) instruction” on page 282

“sync (Synchronize) or dcs (Data Cache Synchronize) instruction” on page 498

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

dclst (Data Cache Line Store) instruction

Purpose

Stores a line of modified data in the data cache into main memory.

Syntax

Bits	Value
0-5	31
6-10	///
11-15	RA
16-20	RB
21-30	630
31	Rc

POWER® family

dclst RA, RB

Description

The **dclst** instruction adds the contents of general-purpose register (GPR) *RA* to the contents of GPR *RB*. It then stores the sum in *RA* as the effective address (EA) if *RA* is not 0 and the instruction does not cause a Data Storage interrupt.

If *RA* is 0, the effective address (EA) is the sum of the contents of GPR *RB* and 0.

Consider the following when using the **dclst** instruction:

- If the line containing the byte addressed by the EA is in the data cache and has been modified, the **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 *RA*.
- Address translation treats the **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 **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

Item	Description
RA	Specifies the source and target general-purpose register where result of operation is stored.
RB	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 concepts:

“clcs (Cache Line Compute Size) instruction” on page 184

“clf (Cache Line Flush) instruction” on page 186

“dcbst (Data Cache Block Store) instruction” on page 208

“dcbt (Data Cache Block Touch) instruction” on page 209

“dcbtst (Data Cache Block Touch for Store) instruction” on page 212

“dcbz or dclz (Data Cache Block Set to Zero) instruction” on page 214

“icbi (Instruction Cache Block Invalidate) instruction” on page 282

“sync (Synchronize) or dcs (Data Cache Synchronize) instruction” on page 498

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

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

Bits	Value
0-5	31
6-10	RT
11-15	RA
16-20	RB
21	OE
22-30	331
31	Rc

POWER® family

div *RT, RA, RB*
div. *RT, RA, RB*
dvo *RT, RA, RB*
dvo. *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*) || (MQ), 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 **div** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
div	0	None	0	None
div.	0	None	1	LT,GT,EQ,SO
dvo	1	SO,OV	0	None
dvo.	1	SO,OV	1	LT,GT,EQ,SO

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

Item Description

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:

```
# 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:

```
# 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:

```
# 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:

```
# 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0-5	31
6-10	D
11-15	A
16-20	B
21	OE
22-30	489
31	Rc

PowerPC64

divd	<i>RT</i> , <i>RA</i> , <i>RB</i> (OE=0 Rc=0)
divd.	<i>RT</i> , <i>RA</i> , <i>RB</i> (OE=0 Rc=1)
divdo	<i>RT</i> , <i>RA</i> , <i>RB</i> (OE=1 Rc=0)
divdo.	<i>RT</i> , <i>RA</i> , <i>RB</i> (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-dividend = (quotient * 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 (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:

Item	Description	
divd	RT,RA,RB	# <i>RT</i> = quotient
mulld	RT,RT,RB	# <i>RT</i> = quotient * divisor
subf	RT,RT,RA	# <i>RT</i> = remainder

Parameters

Item	Description
<i>RT</i>	Specifies target general-purpose register for the result of the computation.
<i>RA</i>	Specifies source general-purpose register for the dividend.
<i>RB</i>	Specifies source general-purpose register for the divisor.

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

Bits	Value
0-5	31
6-10	D
11-15	A
16-20	B
21	OE
22-30	457
31	Rc

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 $0 \leq r < \text{divisor}$.

If an attempt is made to perform the division (*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:

Item	Description	
divdu	RT,RA,RB	# <i>RT</i> = quotient
mulld	RT,RT,RB	# <i>RT</i> = quotient * divisor
subf	RT,RT,RA	# <i>RT</i> = 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

Item	Description
<i>RT</i>	Specifies target general-purpose register for the result of the computation.
<i>RA</i>	Specifies source general-purpose register for the dividend.
<i>RB</i>	Specifies source general-purpose register for the divisor.

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

Bits	Value
0-5	31
6-10	<i>RT</i>
11-15	<i>RA</i>
16-20	<i>RB</i>
21	
22-30	363
31	<i>Rc</i>

POWER® family

divs	<i>RT, RA, RB</i>
divs.	<i>RT, RA, RB</i>
divso	<i>RT, RA, RB</i>
divso.	<i>RT, RA, RB</i>

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 **divs** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
divs	0	None	0	None
divs.	0	None	1	LT,GT,EQ,SO
divso	1	SO,OV	0	None
divso.	1	SO,OV	1	LT,GT,EQ,SO

The four syntax forms of the **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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>RB</i>	Specifies source general-purpose register for operation.

Examples

- 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.
```
- 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.
```
- 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.
```
- 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0-5	31
6-10	RT
11-15	RA
16-20	RB
21	OE
22-30	491
31	Rc

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
divw	0	None	0	None
divw.	0	None	1	LT,GT,EQ,SO
divwo	1	SO, OV	0	None
divwo.	1	SO, OV	1	LT,GT,EQ,SO

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for dividend.
<i>RB</i>	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 0x8000 0000.  
# Assume GPR 6 contains 0xFFFF FFFF.  
divwo. 4,4,6  
# GPR 4 now contains undefined quantity.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0-5	31
6-10	RT
11-15	RA
16-20	RB
21	OE
22-30	459
31	Rc

PowerPC®

divwu *RT, RA, RB*
divwu. *RT, RA, RB*
divwuo *RT, RA, RB*
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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
divwu	0	None	0	None
divwu.	0	None	1	LT,GT,EQ,SO
divwuo	1	SO, OV,	0	None
divwuo.	1	SO, OV	1	LT,GT,EQ,SO

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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.
divwu. 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.
divwuo 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 0x8000 0000.
# Assume GPR 6 contains 0x0000 0002.
divwuo. 4,4,6
# GPR 4 now contains 0x4000 0000.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0-5	31
6-10	RT
11-15	RA
16-20	RB
21	OE
22-30	264
31	Rc

POWER® family

doz *RT, RA, RB*

doz. *RT, RA, RB*

dozo *RT, RA, RB*

dozo. *RT, RA, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
doz	0	None	0	None
doz.	0	None	1	LT,GT,EQ,SO
dozo	1	SO,OV	0	None
dozo.	1	SO,OV	1	LT,GT,EQ,SO

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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:

```
# 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:

```
# 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:

```
# Assume GPR 4 holds 0x0000 0002.
# Assume GPR 6 holds 0x0000 0008.
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:

```
# Assume GPR 4 holds 0xEFFF FFFF.
# Assume GPR 6 holds 0x0000 0000.
dozo. 4,4,6
# GPR 4 now holds 0x1000 0001.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0-5	09
6-10	RT
11-15	RA
16-31	SI

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>SI</i>	Specifies signed 16-bit integer for operation.

The following code determines the difference between GPR 4 and 0x0 and stores the result in GPR 4:

```
# Assume GPR 4 holds 0x0000 0001.
dozi 4,4,0x0
# GPR 4 now holds 0x0000 0000.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0-5	31
6-10	RT
11-15	RA
16-20	RB
21-30	310
31	/

Item	Description
eciwx	<i>RT, RA, RB</i>

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*.

Note:

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>RB</i>	Specifies source general-purpose register for operation.

Related concepts:

“ecowx (External Control Out Word Indexed) instruction”

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

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

Bits	Value
0-5	31
6-10	RS
11-15	RA
16-20	RB
21-30	438
31	/

Item	Description
ecowx	<i>RS, RA, RB</i>

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

Item	Description
<i>RS</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>RB</i>	Specifies source general-purpose register for operation.

Related concepts:

“eciwx (External Control In Word Indexed) instruction” on page 230

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

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

Bits	Value
0-5	31
6-10	///
11-15	///
16-20	///
21-30	854
31	/

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:

```
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 concepts:

“sync (Synchronize) or dcs (Data Cache Synchronize) instruction” on page 498

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

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 doubleword in size (64 bits).

Syntax

Bits	Value
0-5	31
6-10	S
11-15	A
16-20	00000
21-30	986
31	Rc

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

Item	Description
<i>RA</i>	Specifies the target general purpose register for the result of the operation.
<i>RS</i>	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

Bits	Value
0-5	31
6-10	RS
11-15	RA
16-20	RB
21-30	284
31	Rc

Item	Description
eqv	<i>RA, RS, RB</i>
eqv.	<i>RA, RS, RB</i>

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
eqv	None	None	0	None
eqv.	None	None	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>RB</i>	Specifies source general-purpose register for operation.

Examples

- 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.
```
- 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

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

Bits	Value
0-5	31
6-10	RS
11-15	RA
16-20	///
21-30	954
31	Rc

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

Item	Description

<i>RA</i>	Specifies target general-purpose register where result of operation is stored.

<i>RS</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

extsh or exts (Extend Sign Halfword) instruction

Purpose

Extends the lower 16-bit contents of a general-purpose register.

Syntax

Bits	Value
0-5	31
6-10	RS
11-15	RA
16-20	///
21	OE
22-30	922
31	Rc

PowerPC®

```
extsh      RA, RS
extsh.     RA, RS
```

POWER® family

```
exts      RA, RS
exts.     RA, RS
```

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
extsh	None	None	0	None
extsh.	None	None	1	LT,GT,EQ,SO
exts	None	None	0	None
exts.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **extsh** instruction, and the two syntax forms of the **extsh** 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

Item Description

- RA* Specifies general-purpose register receives extended integer.
- RS* Specifies source general-purpose register for operation.

Examples

- 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 0xFFFF FFFF.
```
- 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

fabs (Floating Absolute Value) instruction

Purpose

Stores the absolute value of the contents of a floating-point register in another floating-point register.

Syntax

Bits	Value
0-5	63
6-10	FRT
11-15	///
16-20	FRB
21-30	264
31	Rc

Item	Description
fabs	<i>FRT, FRB</i>
fabs.	<i>FRT, FRB</i>

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fabs	None	0	None
fabs.	None	1	FX,FEX,VX,OX

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

Item	Description
<i>FRT</i>	Specifies target floating-point register for operation.
<i>FRB</i>	Specifies source floating-point register for operation.

Examples

- The following code sets bit 0 of FPR 4 to zero and place sthe result in FPR 6:


```
# Assume FPR 4 holds 0xC053 4000 0000 0000.
fabs 6,4
# GPR 6 now holds 0x4053 4000 0000 0000.
```
- 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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point move instructions” on page 30

The Floating-point move instructions copy data from one FPR to another FPR.

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

fadd or fa (Floating Add) instruction

Purpose

Adds two floating-point operands and places the result in a floating-point register.

Syntax

Bits	Value
0-5	63
6-10	FRT
11-15	FRA
16-20	FRB
21-25	///
26-30	21
31	Rc

PowerPC®

fadd *FRT, FRA, FRB*

fadd. *FRT, FRA, FRB*

POWER® family

fa *FRT, FRA, FRB*

fa. *FRT, FRA, FRB*

Bits	Value
0-5	59
6-10	FRT
11-15	FRA
16-20	FRB
21-25	///
26-30	21
31	Rc

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fadd	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	0	None
fadd.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	1	FX,FEX,VX,OX
fadds	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	0	None
fadds.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	1	FX,FEX,VX,OX
fa	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	0	None
fa.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	1	FX,FEX,VX,OX

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

Item	Description
<i>FRT</i>	Specifies target floating-point register for operation.
<i>FRA</i>	Specifies source floating-point register for operation.
<i>FRB</i>	Specifies source floating-point register for operation.

Examples

- 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.
```
- 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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point arithmetic instructions” on page 30

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

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

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

Bits	Value
0-5	63
6-10	D
11-15	00000
16-20	B
21-30	846
31	Rc

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fcfid	FPRE,FR,FI,FX,XX	0	None
fcfid.	FPRE,FR,FI,FX,XX	1	FX,FEX,VX,OX

Parameters

Item	Description
<i>FRT</i>	Specifies the target floating-point register for the operation.
<i>FRB</i>	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

Bits	Value
0-5	63
6-8	BF
9-10	//
11-15	FRA
16-20	FRB
21-30	32
31	/

Item	Description
fcmpo	<i>BF, FRA, FRB</i>

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 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

Item	Description
<i>BF</i>	Specifies field in the condition register that receives the four FPCC bits.
<i>FRA</i>	Specifies source floating-point register.
<i>FRB</i>	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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point compare instructions” on page 31

The Floating-point compare instructions perform ordered and unordered comparisons of the contents of two FPRs.

fcmpu (Floating Compare Unordered) instruction

Purpose

Compares the contents of two floating-point registers.

Syntax

Bits	Value
0-5	63
6-8	BF
9-10	//
11-15	FRA
16-20	FRB
21-30	0
31	/

Item	Description
fcmpu	<i>BF, FRA, FRB</i>

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

Item Description

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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point compare instructions” on page 31

The Floating-point compare instructions perform ordered and unordered comparisons of the contents of two FPRs.

ftcid (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

Bits	Value
0-5	63
6-10	D
11-15	00000
16-20	B
21-30	814
31	Rc

PowerPC®

ftid *FRT, FRB* (Rc=0)
ftid. *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 **ftid** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
ftid	FPRF(undefined),FR,FI,FX,XX,VXSNAN,VXCVI	0	None
ftid.	FPRF(undefined),FR,FI,FX,XX,VXSNAN,VXCVI	1	FX,FEX,VX,OX

Parameters

Item Description

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.

ftidz (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

Bits	Value
0-5	63
6-10	D
11-15	00000
16-20	B
21-30	815
31	Rc

PowerPC®

fctidz *FRT, FRB* (Rc=0)

fctidz. *FRT, FRB* (Rc=1)

Description

The floating-point operand in floating-point register (FRP) *FRB* is converted to a 64-bit signed fixed-point integer, using the rounding mode round toward zero, and placed into FPR *FRT*.

If the operand in FPR *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 **fctidz** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fctidz	FPRF(undefined),FR,FI,FX,XX,VXSNAN,VXCVI	0	None
fctidz.	FPRF(undefined),FR,FI,FX,XX,VXSNAN,VXCVI	1	FX,FEX,VX,OX

Parameters

Item Description

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

Bits	Value
0-5	63
6-10	FRT
11-15	///
16-20	FRB
21-30	14
31	Rc

PowerPC®

fctiw *FRT, FRB*

fctiw. *FRT, FRB*

POWER2™

fcir *FRT, FRB*

fcir. *FRT, FRB*

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fctiw	C,FL,FG,FE,FU,FR,FI,FX,XX,VXCVI, VXSNNAN	0	None
fctiw.	C,FL,FG,FE,FU,FR,FI,FX,XX,VXCVI, VXSNNAN	1	FX,FEX,VX,OX
fcir	C,FL,FG,FE,FU,FR,FI,FX,XX,VXCVI, VXSNNAN	0	None
fcir.	C,FL,FG,FE,FU,FR,FI,FX,XX,VXCVI, VXSNNAN	1	FX,FEX,VX,OX

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

Item	Description
<i>FRT</i>	Specifies the floating-point register where the integer result is placed.
<i>FRB</i>	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:

```
# 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.
lwz    3,TEMP+4(1)       # Get the integer part of the
                        # doubleword.
lfd    5,0(3)            # Get the selected array element.
# FPR 5 now contains the selected array element.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point arithmetic instructions” on page 30

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

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

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

Bits	Value
0-5	63
6-10	FRT
11-15	///
16-20	FRB
21-30	15
31	Rc

PowerPC®

fctiwz *FRT, FRB*
fctiwz. *FRT, FRB*

POWER2™

fcirz *FRT, FRB*
fcirz. *FRT, FRB*

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fctiwz	C,FL,FG,FE,FU,FR,FI,FX,XX,VXCVI, VXSNaN	0	None
fctiwz.	C,FL,FG,FE,FU,FR,FI,FX,XX,VXCVI, VXSNaN	1	FX,FEX,VX,OX
fcirz	C,FL,FG,FE,FU,FR,FI,FX,XX,VXCVI, VXSNaN	0	None
fcirz.	C,FL,FG,FE,FU,FR,FI,FX,XX,VXCVI, VXSNaN	1	FX,FEX,VX,OX

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

Item Description

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
```

```

lfdx    5,3,4      # doubleword.
fadd    5,5,4      # Get the selected array element.
stfd    5,3,4      # Add value1 to array element.
          # Save the new value of the
          # array element.

```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point arithmetic instructions” on page 30

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

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

fddiv or fd (Floating Divide) instruction

Purpose

Divides one floating-point operand by another.

Syntax

Bits	Value
0-5	63
6-10	FRT
11-15	FRA
16-20	FRB
21-25	///
26-30	18
31	Rc

PowerPC®

fddiv *FRT, FRA, FRB*

fddiv. *FRT, FRA, FRB*

POWER® family

fd *FRT, FRA, FRB*

fd. *FRT, FRA, FRB*

Bits	Value
0-5	59
6-10	FRT
11-15	FRA
16-20	FRB
21-25	///
26-30	18
31	Rc

fdivs *FRT, FRA, FRB*
fdivs. *FRT, FRA, FRB*

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fdiv	C,FL,FG,FE,FU,FR,FI,OX,UX, ZX,XX,VXSNAN,VXIDI,VXZDZ	0	None
fdiv.	C,FL,FG,FE,FU,FR,FI,OX,UX, ZX,XX,VXSNAN,VXIDI,VXZDZ	1	FX,FEX,VX,OX
fdivs	C,FL,FG,FE,FU,FR,FI,OX,UX, ZX,XX,VXSNAN,VXIDI,VXZDZ	0	None
fdivs.	C,FL,FG,FE,FU,FR,FI,OX,UX, ZX,XX,VXSNAN,VXIDI,VXZDZ	1	FX,FEX,VX,OX
fd	C,FL,FG,FE,FU,FR,FI,OX,UX, ZX,XX,VXSNAN,VXIDI,VXZDZ	0	None
fd.	C,FL,FG,FE,FU,FR,FI,OX,UX, ZX,XX,VXSNAN,VXIDI,VXZDZ	1	FX,FEX,VX,OX

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

Item	Description
<i>FRT</i>	Specifies target floating-point register for operation.
<i>FRA</i>	Specifies source floating-point register containing the dividend.
<i>FRB</i>	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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point arithmetic instructions” on page 30

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

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

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

Bits	Value
0-5	63
6-10	FRT
11-15	FRA
16-20	FRB
21-25	FRC
26-30	29
31	Rc

PowerPC®

fmadd *FRT, FRA, FRC, FRB*
fmadd. *FRT, FRA, FRC, FRB*

POWER® family

fma *FRT, FRA, FRC, FRB*
fma. *FRT, FRA, FRC, FRB*

Bits	Value
0-5	59
6-10	FRT
11-15	FRA
16-20	FRB
21-25	FRC
26-30	29
31	Rc

PowerPC®

fmadds *FRT, FRA, FRC, FRB*
fmadds. *FRT, FRA, FRC, FRB*

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fmadd	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	0	None
fmadd.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	1	FX,FEX,VX,OX
fmadds	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	0	None
fmadds.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	1	FX,FEX,VX,OX
fma	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	0	None
fma.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	1	FX,FEX,VX,OX

All syntax forms of the **fmadd**, **fmadds**, and **fma** 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

Item Description

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:

```
# 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 D7FF 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:

```
# 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.
fmadd. 6,4,5,7
# FPR 6 now contains 0xC070 D7FF FFFF F6CB.
# FPSCR now contains 0x8206 8000.
# CR now contains 0x0800 0000.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

fmr (Floating Move Register) instruction

Purpose

Copies the contents of one floating-point register into another floating-point register.

Syntax

Bits	Value
0-5	63
6-10	FRT
11-15	///
16-20	FRB
21-30	72
31	Rc

Item	Description
fmr	<i>FRT, FRB</i>
fmr.	<i>FRT, FRB</i>

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fmr	None	0	None
fmr.	None	1	FX,FEX,VX,OX

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

Item	Description
<i>FRT</i>	Specifies target floating-point register for operation.
<i>FRB</i>	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:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.  
# Assume FPSCR = 0.  
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.
```

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

Bits	Value
0-5	63
6-10	FRT
11-15	FRA
16-20	FRB
21-25	FRC
26-30	28
31	Rc

PowerPC®

fmsub *FRT, FRA, FRC, FRB*

fmsub. *FRT, FRA, FRC, FRB*

POWER® family

fms *FRT, FRA, FRC, FRB*

fms. *FRT, FRA, FRC, FRB*

Bits	Value
0-5	59
6-10	FRT
11-15	FRA
16-20	FRB
21-25	FRC
26-30	28
31	Rc

fmsubs *FRT, FRA, FRC, FRB***fmsubs.** *FRT, FRA, FRC, FRB*

Description

The **fmsub** and **fms** 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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fmsub	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXSI,VXIMZ	0	None
fmsub.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXSI,VXIMZ	1	FX,FEX,VX,OX
fmsubs	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXSI,VXIMZ	0	None
fmsubs.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXSI,VXIMZ	1	FX,FEX,VX,OX
fms	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXSI,VXIMZ	0	None
fms.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXSI,VXIMZ	1	FX,FEX,VX,OX

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

Item	Description
<i>FRT</i>	Specifies target floating-point register for operation.
<i>FRA</i>	Specifies source floating-point register containing a multiplier.
<i>FRB</i>	Specifies source floating-point register containing the quantity to be subtracted.
<i>FRC</i>	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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

“clcs (Cache Line Compute Size) instruction” on page 184

“clf (Cache Line Flush) instruction” on page 186

“cli (Cache Line Invalidate) instruction” on page 187

“dcbf (Data Cache Block Flush) instruction” on page 205

“dcbst (Data Cache Block Store) instruction” on page 208

“dcbt (Data Cache Block Touch) instruction” on page 209

“dcbtst (Data Cache Block Touch for Store) instruction” on page 212

“icbi (Instruction Cache Block Invalidate) instruction” on page 282

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

fmul or fm (Floating Multiply) instruction

Purpose

Multiplies two floating-point operands.

Syntax

Bits	Value
0-5	63
6-10	FRT
11-15	FRA
16-20	///
21-25	FRC
26-30	25
31	Rc

PowerPC®

fmul *FRT, FRA, FRC*

fmul. *FRT, FRA, FRC*

POWER® family

fm *FRT, FRA, FRC*

fm. *FRT, FRA, FRC*

Bits	Value
0-5	59
6-10	FRT
11-15	FRA
16-20	///
21-25	FRC
26-30	25
31	Rc

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fmul	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXIMZ	0	None
fmul.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXIMZ	1	FX,FEX,VX,OX
fmuls	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXIMZ	0	None
fmuls.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXIMZ	1	FX,FEX,VX,OX
fm	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXIMZ	0	None
fm.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXIMZ	1	FX,FEX,VX,OX

All syntax forms of the **fmul**, **fmuls**, 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

Item	Description
<i>FRT</i>	Specifies target floating-point register for operation.
<i>FRA</i>	Specifies source floating-point register for operation.
<i>FRC</i>	Specifies source floating-point register for operation.

Examples

- 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.
```
- 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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point arithmetic instructions” on page 30

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

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

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

Bits	Value
0-5	63
6-10	FRT
11-15	///
16-20	FRB
21-30	136
31	/

Item	Description
fnabs	<i>FRT</i> , <i>FRB</i>
fnabs.	<i>FRT</i> , <i>FRB</i>

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fnabs	None	0	None
fnabs.	None	1	FX,FEX,VX,OX

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

Item	Description
<i>FRT</i>	Specifies target floating-point register for operation.
<i>FRB</i>	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.
```

fneg (Floating Negate) instruction

Purpose

Negates the contents of a floating-point register and places the result into another floating-point register.

Syntax

Bits	Value
0-5	63
6-10	FRT
11-15	///
16-20	FRB
21-30	40
31	Rc

Item	Description
fneg	<i>FRT, FRB</i>
fneg.	<i>FRT, FRB</i>

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fneg	None	0	None
fneg.	None	1	FX,FEX,VX,OX

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

Item	Description
<i>FRT</i>	Specifies target floating-point register for operation.
<i>FRB</i>	Specifies source floating-point register for operation.

Examples

- 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.
```
- 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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point move instructions” on page 30

The Floating-point move instructions copy data from one FPR to another FPR.

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

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

Bits	Value
0-5	63
6-10	FRT
11-15	FRA
16-20	FRB
21-25	FRC
26-30	31
31	Rc

PowerPC®

fnmadd *FRT, FRA, FRC, FRB*
fnmadd. *FRT, FRA, FRC, FRB*

POWER® family

fnma *FRT, FRA, FRC, FRB*
fnma. *FRT, FRA, FRC, FRB*

Bits	Value
0-5	59
6-10	FRT
11-15	FRA
16-20	FRB
21-25	FRC
26-30	31
31	Rc

PowerPC®

fnmadds *FRT, FRA, FRC, FRB*
fnmadds. *FRT, FRA, FRC, FRB*

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fnmadd	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	0	None
fnmadd.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	1	FX,FEX,VX,OX
fnmadds	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	0	None
fnmadds.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	1	FX,FEX,VX,OX
fnma	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	0	None
fnma.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	1	FX,FEX,VX,OX

All syntax forms of the **fnmadd**, **fnmadds**, and **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.

Note: Rounding occurs before the result of the addition is negated. Depending on *RN*, an inexact value may result.

Parameters

Item	Description
<i>FRT</i>	Specifies target floating-point register for operation.
<i>FRA</i>	Specifies source floating-point register for operation.
<i>FRB</i>	Specifies source floating-point register for operation.
<i>FRC</i>	Specifies source floating-point register for operation.

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:

```
# 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.
```

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:

```
# 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.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the

instruction.

fnmsub or fnms (Floating Negative Multiply-Subtract) instruction

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

Bits	Value
0-5	63
6-10	FRT
11-15	FRA
16-20	FRB
21-25	FRC
26-30	30
31	Rc

PowerPC®

fnmsub *FRT, FRA, FRC, FRB*

fnmsub. *FRT, FRA, FRC, FRB*

POWER® family

fnms *FRT, FRA, FRC, FRB*

fnms. *FRT, FRA, FRC, FRB*

Bits	Value
0-5	59
6-10	FRT
11-15	FRA
16-20	FRB
21-25	FRC
26-30	
30	Rc

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fnmsub	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	0	None
fnmsub.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	1	FX,FEX,VX,OX
fnmsubs	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	0	None
fnmsubs.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	1	FX,FEX,VX,OX
fnms	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	0	None
fnms.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	1	FX,FEX,VX,OX

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

Item	Description
<i>FRT</i>	Specifies target floating-point register for operation.
<i>FRA</i>	Specifies first source floating-point register for operation.
<i>FRB</i>	Specifies second source floating-point register for operation.
<i>FRC</i>	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:

```
# 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.
fnmsub 6,4,5,7
# FPR 6 now contains 0x4070 D800 0000 0935.
# FPSCR now contains 0x8202 4000.
```

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 6AB4 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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

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

Bits	Value
0-5	59
6-10	FRT
11-15	///
16-20	FRB
21-25	///
26-30	24
31	Rc

PowerPC®

fres *FRT, FRB*

fres. *FRT, FRB*

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 *FRT* is correct to a precision of one part in 256 of the reciprocal of *FRB*. The value placed into *FRT* may vary between implementations, and between different executions on the same implementation.

The following table summarizes special conditions:

Item	Description	
Special Conditions		
Operand	Result	Exception
Negative Infinity	Negative 0	None
Negative 0	Negative Infinity ¹	ZX
Positive 0	Positive Infinity ¹	ZX
Positive Infinity	Positive 0	None
SNaN	QNaN ²	VXSNAN
QNaN	QNaN	None

¹No result if FPSCRZE = 1.

²No 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 **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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fres	C,FL,FG,FE,FU,FR,FI,FX,OX, UX,ZX,VXSNAN	0	None
fres.	C,FL,FG,FE,FU,FR,FI,FX,OX, UX,ZX,VXSNAN	1	FX,FEX,VX,OX

The **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 **fres** syntax form sets the Record (Rc) bit to 0 and does not affect Condition Register Field 1 (CR1).

Parameters

Item Description

FRT Specifies target floating-point register for operation.

FRB Specifies source floating-point register for operation.

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point arithmetic instructions” on page 30

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

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

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

Bits	Value
0-5	63
6-10	FRT
11-15	///
16-20	FRB
21-30	12
31	Rc

Item	Description
frsp	<i>FRT, FRB</i>
frsp.	<i>FRT, FRB</i>

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
frsp	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN	0	None
frsp.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN	1	FX,FEX,VX,OX

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

Item	Description
<i>FRT</i>	Specifies target floating-point register for operation.
<i>FRB</i>	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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

“Floating-point arithmetic instructions” on page 30

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

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

Bits	Value
0-5	63
6-10	FRT
11-15	///
16-20	FRB
21-25	///
26-30	26
31	Rc

PowerPC®

frsqrte *FRT, FRB*

frsqrte. *FRT, FRB*

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:

Item	Description	
Special Conditions		
Operand	Result	Exception
Negative Infinity	QNaN ¹	VXSQRT
Less Than 0	QNaN ¹	VXSQRT
Negative 0	Negative Infinity ²	ZX
Positive 0	Positive Infinity ²	ZX
Positive Infinity	Positive 0	None
SNaN	QNaN ¹	VXSNAN
QNaN	QNaN	None

¹No 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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
frsqrte	C,FL,FG,FE,FU,FR,FI,FX,ZX, VXSNaN,VXSQRT	0	None
frsqrte.	C,FL,FG,FE,FU,FR,FI,FX,ZX, VXSNaN,VXSQRT	1	FX,FEX,VX,OX

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

Item Description

FRT Specifies target floating-point register for operation.
FRB Specifies source floating-point register for operation.

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point arithmetic instructions” on page 30

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

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

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 **fsel** 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

Bits	Value
0-5	63
6-10	FRT
11-15	FRA
16-20	FRB
21-25	FRC
26-30	23
31	Rc

fsel *FRT, FRA, FRC, FRB***fsel.** *FRT, FRA, FRC, FRB*

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 **fsel** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Item	Description		
Syntax Form	FPSCR bits	Record Bit (Rc)	Condition Register Field 1
fsel	None	0	None
fsel.	None	1	FX, FEX, VX, OX

The two syntax forms of the **fsel** 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

Item Description

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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

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

Bits	Value
0-5	63
6-10	D
11-15	00000
16-20	B
21-25	00000
26-30	22
31	Rc

PowerPC®

fsqrt *FRT, FRB (Rc=0)*

fsqrt. *FRT, FRB (Rc=1)*

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.

Operand	Result	Exception
- infinity	QNaN*	VXSQRT
< 0	QNaN*	VXSQRT
- 0	- 0	None
+ infinity	+ infinity	None
SNaN	QNaN*	VXSNAN
QNaN	QNaN	None

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fsqrt	FPRE,FR,FI,FX,XX,VXSNAN,VXSQRT	0	None
fsqrt.	FPRE,FR,FI,FX,XX,VXSNAN,VXSQRT	1	FX,FEX,VX,OX

Parameters

Item	Description
<i>FRT</i>	Specifies the target floating-point register for the operation.
<i>FRB</i>	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

Bits	Value
0-5	59
6-10	D
11-15	00000
16-20	B
21-25	00000
26-30	22
31	Rc

PowerPC®

fsqrts	<i>FRT, FRB</i> (Rc=0)
fsqrts.	<i>FRT, FRB</i> (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.

Operand	Result	Exception
- infinity	QNaN*	VXSQRT
< 0	QNaN*	VXSQRT
- 0	- 0	None
+ infinity	+ infinity	None
SNaN	QNaN*	VXSNAN
QNaN	QNaN	None

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fsqrts	FPRF,FR,FI,FX,XX,VXSNAN,VXSQRT	0	None
fsqrts.	FPRF,FR,FI,FX,XX,VXSNAN,VXSQRT	1	FX,FEX,VX,OX

Parameters

Item	Description
<i>FRT</i>	Specifies the target floating-point register for the operation.
<i>FRB</i>	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

Bits	Value
0-5	63
6-10	FRT
11-15	FRA
16-20	FRB
21-25	///
26-30	20
31	Rc

PowerPC®

fsub *FRT, FRA, FRB*
fsub. *FRT, FRA, FRB*

PowerPC®

fs *FRT, FRA, FRB*
fs. *FRT, FRA, FRB*

Bits	Value
0-5	59
6-10	FRT
11-15	FRA
16-20	FRB
21-25	///
26-30	20
31	Rc

PowerPC®

fsubs *FRT, FRA, FRB*
fsubs. *FRT, FRA, FRB*

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.

Item	Description		
Syntax Form	Floating-Point Status and Control Register	Record Bit (Rc)	Condition Register Field 1
fsub	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	0	None
fsub.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	1	FX,FEX,VX,OX
fsubs	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	0	None
fsubs.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	1	FX,FEX,VX,OX
fs	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	0	None
fs.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	1	FX,FEX,VX,OX

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

Item Description

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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point arithmetic instructions” on page 30

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

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

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

Bits	Value
0-5	31
6-10	///
11-15	RA
16-20	RB
21-30	982
31	/

PowerPC®

icbi *RA, RB*

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

Item	Description

<i>RA</i>	Specifies source general-purpose register for the EA calculation.

<i>RB</i>	Specifies source general-purpose register for the EA calculation.

Examples

The following code ensures that modified instructions are available for execution:

```
# Assume GPR 3 contains a modified instruction.
# Assume GPR 4 contains the address of the memory location
# where the modified instruction will be stored.
stw    3,0(4)      # Store the modified instruction.
dcbf   0,4          # Copy the modified instruction to
```

```

sync                # main memory.
                    # Ensure update is in main memory.
icbi    0,4         # Invalidate block with old instruction.
isync              # Discard prefetched instructions.
b        newcode    # Go execute the new code.

```

Related concepts:

“clcs (Cache Line Compute Size) instruction” on page 184

“clf (Cache Line Flush) instruction” on page 186

“dcbf (Data Cache Block Flush) instruction” on page 205

“dcbst (Data Cache Block Store) instruction” on page 208

“dcbt (Data Cache Block Touch) instruction” on page 209

“dcbtst (Data Cache Block Touch for Store) instruction” on page 212

“dcbz or dclz (Data Cache Block Set to Zero) instruction” on page 214

“icbi (Instruction Cache Block Invalidate) instruction” on page 282

“sync (Synchronize) or dcs (Data Cache Synchronize) instruction” on page 498

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

isync or ics (Instruction Synchronize) instruction

Purpose

Refetches any instructions that might have been fetched prior to this instruction.

Syntax

Bits	Value
0-5	19
6-10	///
11-15	///
16-20	///
21-30	150
31	/

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 concepts:

“clcs (Cache Line Compute Size) instruction” on page 184

“clf (Cache Line Flush) instruction” on page 186

“cli (Cache Line Invalidate) instruction” on page 187

“dcbf (Data Cache Block Flush) instruction” on page 205

“dcbi (Data Cache Block Invalidate) instruction” on page 206

“dcbz or dclz (Data Cache Block Set to Zero) instruction” on page 214

“dclst (Data Cache Line Store) instruction” on page 216

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“Functional differences for POWER[®] family and PowerPC[®] instructions” on page 144

The POWER[®] family and PowerPC[®] instructions that share the same op code on POWER[®] family and PowerPC[®] platforms, but differ in their functional definition.

lbz (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

Bits	Value
0-5	34
6-10	RT
11-15	RA
16-31	D

Item	Description
lbz	<i>RT</i> , <i>D</i> (<i>RA</i>)

Description

The **lbz** 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 **lbz** instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>D</i>	16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

lbzu (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

Bits	Value
0-5	35
6-10	<i>RT</i>
11-15	<i>RA</i>
16-31	<i>D</i>

Item	Description
lbzu	<i>RT</i> , <i>D</i> (<i>RA</i>)

Description

The **lbzu** 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 **lbzu** instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>D</i>	16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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:

```
.csect data[rw]
storage: .byte 0x61
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
lbzu 6,storage(5)
# GPR 6 now contains 0x0000 0061.
# GPR 5 now contains the storage address.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

lbzux (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

Bits	Value
0-5	31
6-10	RT
11-15	RA
16-20	RB
21-30	119
31	/

Item	Description
lbzux	<i>RT, RA, RB</i>

Description

The **lbzux** 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 **lbzux** instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for EA calculation and possible address update.
<i>RB</i>	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:

```
storage: .byte 0x40
.
.
# Assume GPR 5 contains 0x0000 0000.
# Assume GPR 4 is the storage address.
lbzux 6,5,4
# GPR 6 now contains 0x0000 0040.
# GPR 5 now contains the storage address.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

lbzx (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

Bits	Value
0-5	31
6-10	RT
11-15	RA
16-20	RB
21-30	87
31	/

Item	Description
lbzx	RT, RA, RB

Description

The **lbzx** 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 **lbzx** instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters

Item	Description
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:

```
storage: .byte 0x61
        .
        .
# Assume GPR 5 contains 0x0000 0000.
# Assume GPR 4 is the storage address.
lbzx 6,5,4
# GPR 6 now contains 0x0000 0061.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

ld (Load Doubleword) instruction

Purpose

Load a doubleword 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

Bits	Value
0 - 5	58
6 - 10	RT
11 - 15	RA
16 - 29	DS
30 - 31	0b00

PowerPC 64

ld *RT, Disp(RA)*

Description

The **ld** instruction loads a doubleword in storage from a specified location in memory addressed by the effective address (EA) into the target general-purpose register (GPR) *RT*.

DS is a 14-bit, signed two's complement number, which is sign-extended to 64 bits, and then multiplied by 4 to provide a displacement *Disp*. If GPR *RA* is not 0, the EA is the sum of the contents of GPR *RA* and *Disp*. If GPR *RA* is 0, then the EA is *Disp*.

Parameters

Item	Description

<i>RT</i>	Specifies target general-purpose register where result of operation is stored.

<i>Disp</i>	Specifies a 16-bit signed number that is a multiple of 4. The assembler divides this number by 4 when generating the instruction.

<i>RA</i>	Specifies source general-purpose register for EA calculation.

Examples

The following code loads a doubleword from memory into GPR 4:

```
.extern mydata[RW]
.csect foodata[RW]
.local foodata[RW]
storage: .llong mydata # address of mydata

.csect text[PR]
ld 4,storage(5) # Assume GPR 5 contains address of csect foodata[RW].
# GPR 4 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

ldarx (Load Doubleword Reserve Indexed) instruction

Purpose

The **ldarx** instruction is used in conjunction with a subsequent **stdcx** instruction to emulate a read-modify-write operation on a specified memory location.

Syntax

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21 - 30	84
31	/

PowerPC64

ldarx *RT, RA, RB*

Description

The **ldarx** and **stdcx** (**Store Doubleword Conditional Indexed**) instructions are used to perform a read-modify-write operation to storage. If the store operation is performed, the use of the **ldarx** and **stdcx** instructions ensures that no other processor or mechanism changes the target memory location between the time the **ldarx** instruction is run and the time the **stdcx** instruction is completed.

If general-purpose register (GPR) *RA* equals 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 **ldarx** instruction loads the word from the location in storage that is 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 **ldarx** instruction has one syntax form and does not affect the Fixed-Point Exception Register. If the EA is not a multiple of 8, either the system alignment handler is invoked or the results are called undefined.

Parameters

Item	Description
RT	Specifies the source GPR of the stored data.
RA	Specifies the source GPR for the EA calculation.
RB	Specifies the source GPR for the EA calculation.

Examples

1. The following code performs a fetch and store operation 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 operation 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.

Idu (Load Doubleword with Update) instruction

Purpose

Loads a doubleword of data into the specified general purpose register (GPR) , and updates the address base.

Note: This instruction should only be used on 64-bit PowerPC processors that run a 64-bit application.

Syntax

Bits	Value
0 - 5	58
6 - 10	RT
11 - 15	RA
16 - 29	DS
30 - 31	0b01

PowerPC 64

ldu *RT, Disp(RA)*

Description

The **ldu** instruction loads a doubleword in storage from a specified location in memory that is addressed by the effective address (EA) into the target GPR *RT*.

DS is a 14-bit, signed two's complement number, which is sign-extended to 64 bits, and then multiplied by 4 to provide a displacement (*Disp*). If GPR *RA* is not 0, the EA is the sum of the contents of GPR *RA* and *Disp*.

If *RA* equals 0 or *RA* equals *RT*, the instruction form is invalid.

Parameters

Item Description

RT Specifies the target GPR where the result of the operation is stored.

Disp Specifies a 16-bit signed number that is a multiple of 4. The assembler divides this number by 4 when generating the instruction.

RA Specifies the source GPR for the EA calculation.

Examples

The following code loads the first of four doublewords from memory into GPR 4, and increments to GPR 5 to point to the next doubleword in memory:

```
.csect    foodata[RW]
storage: .llong 5,6,7,12 # Successive doublewords.

.csect    text[PR]
ldu      4,storage(5)    # Assume GPR 5 contains address of csect foodata[RW].
                        # GPR 4 now contains the first doubleword of
                        # foodata; GRP 5 points to the second doubleword.
```

Implementation

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation invokes the system illegal instruction error handler.

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

ldux (Load Doubleword with Update Indexed) instruction

Purpose

Loads a doubleword of data from a specified memory location into a general purpose register (GPR), and updates the address base.

Syntax

Bits	Value
0 - 5	31
6 - 10	D
11 - 15	A
16 - 20	B
21 - 30	53
31	0

PowerPC®

ldux *RT, RA, RB*

Description

The effective address (EA) is calculated from the sum of the GPR, *RA* and *RB*. A doubleword of data is read from the memory location that is referenced by the EA and placed into GPR *RT*. GPR *RA* is updated with the EA.

If *RA* equals 0 or *RA* equals *RD*, the instruction form is invalid.

Parameters

Item	Description
<i>RT</i>	Specifies the source GPR for the stored data.
<i>RA</i>	Specifies source GPR for the EA calculation.
<i>RB</i>	Specifies source GPR for the EA calculation.

Implementation

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation calls the system illegal instruction error handler.

ldx (Load Doubleword Indexed) instruction

Purpose

Loads a doubleword from a specified memory location into a general purpose register (GPR).

Syntax

Bits	Value
0 - 5	31
6 - 10	D
11 - 15	A
16 - 20	B
21 - 30	21
31	0

PowerPC®

ldx *RT RA, RB*

Description

The **ldx** instruction loads a doubleword from the specified memory location that is referenced by the effective address (EA) into the GPR *RT*.

If GRP *RA* is not 0, the effective address (EA) is the sum of the contents of GRPs, *RA* and *RB*. Otherwise, the EA is equal to the contents of *RB*.

Parameters

Item	Description
<i>RT</i>	Specifies the target GPR where the result of the operation is stored.
<i>RA</i>	Specifies the source GPR for the EA calculation.
<i>RB</i>	Specifies the source GPR for the EA calculation.

Implementation

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation calls the system illegal instruction error handler.

lfd (Load Floating-Point Double) instruction

Purpose

Loads a doubleword of data from a specified location in memory into a floating-point register.

Syntax

Bits	Value
0 - 5	50
6 - 10	FRT
11 - 15	RA
16 - 31	D

Item	Description
lfd	<i>FRT, D(RA)</i>

Description

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

Item	Description
<i>FRT</i>	Specifies target general-purpose register where result of the operation is stored.
<i>D</i>	16-bit, signed two's complement integer sign-extended to 32 bits for the EA calculation.
<i>RA</i>	Specifies source general-purpose register for the EA calculation.

Examples

The following code loads a doubleword from memory into FPR 6:

```
.csect data[rw]
storage: .double 0x1
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
lfd 6,storage(5)
# FPR 6 now contains 0x3FF0 0000 0000 0000.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

lfd (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

Bits	Value
0 - 5	51
6 - 10	FRT
11 - 15	RA
16 - 31	D

Item	Description
lfd	<i>FRT, D(RA)</i>

Description

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 *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 **lfd** instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

Item	Description
<i>FRT</i>	Specifies target general-purpose register where result of operation is stored.
<i>D</i>	Specifies a 16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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]
lfd 6,storage(5)
# FPR 6 now contains 0x3FF0 0000 0000 0000.
# GPR 5 now contains the storage address.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

lfdx (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

Bits	Value
0 - 5	31
6 - 10	FRT
11 -15	RA
16 - 20	RB
21 - 30	631
31	/

Item	Description
lfdux	<i>FRT, RA, RB</i>

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) *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 *RB*.

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 **lfdux** instruction has one syntax form and does not affect the Floating-Point Status and Control Register.

Parameters

Item	Description
<i>FRT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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:

```
.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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

lfdx (Load Floating-Point Double-Indexed) instruction

Purpose

Loads a doubleword of data from a specified location in memory into a floating-point register.

Syntax

Bits	Value
0 - 5	31
6 - 10	FRT
11 - 15	RA
16 - 20	RB
21 - 30	599
31	/

Item	Description
lfdx	<i>FRT, RA, RB</i>

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

Item	Description
<i>FRT</i>	Specifies target floating-point register where data is stored.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	Specifies source general-purpose register for EA calculation.

Examples

The following code loads a doubleword from memory into FPR 6:

```
storage: .double 0x1
        .
        .
# Assume GPR 4 contains the storage address.
lfdx 6,0,4
# FPR 6 now contains 0x3FF0 0000 0000 0000.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

Ifq (Load Floating-Point Quad) instruction

Purpose

Loads two double-precision values into floating-point registers.

Note: The **Ifq** instruction is supported only in the POWER2™ implementation of the POWER® family architecture.

Syntax

Bits	Value
0 - 5	56
6 - 10	FRT
11 - 15	RA
16 - 29	DS
30 - 31	00

POWER2™

Ifq *FRT, DS(RA)*

Description

The **Ifq** 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 **Ifq** instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

Item	Description

<i>FRT</i>	Specifies the first of two target floating-point registers.

<i>DS</i>	Specifies a 14-bit field used as an immediate value for the EA calculation.

<i>RA</i>	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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

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 **Ifqu** instruction is supported only in the POWER2™ implementation of the POWER® family architecture.

Syntax

Bits	Value
0 - 5	57
6 - 10	FRT
11 - 15	RA
16 - 29	DS
30 - 31	00

POWER2™

Ifqu *FRT, DS(RA)*

Description

The **Ifqu** 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.

If GPR *RA* is not 0, the EA is placed into GPR *RA*.

The **Ifqu** instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

Item	Description
<i>FRT</i>	Specifies the first of two target floating-point register.
<i>DS</i>	Specifies a 14-bit field used as an immediate value for the EA calculation.
<i>RA</i>	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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

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

Bits	Value
0 - 5	31
6 - 10	FRT
11 - 15	RA
16 - 20	RB
21 - 30	823
31	Rc

Description

The **lfqux** 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 **lfqux** instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

Item	Description
<i>FRT</i>	Specifies the first of two target floating-point registers.
<i>RA</i>	Specifies the first source general-purpose register for the EA calculation and the target register for the EA update.
<i>RB</i>	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.
lfq    7,0(3)          # Get first pair of X_Y values.
lwz    5,16(3)         # Get the offset to second element.
lfqux  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.
lfqux  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   7,0,4           # Store the two doubleword results.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

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

Bits	Value
0 - 5	31
6 - 10	FRT
11 - 15	RA
16 - 20	RB
21 - 30	791
31	Rc

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

Item	Description

<i>FRT</i>	Specifies the first of two target floating-point registers.

<i>RA</i>	Specifies one source general-purpose register for the EA calculation.

<i>RB</i>	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:

```
# 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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other

operations.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

lfs (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

Bits	Value
0 - 5	48
6 - 10	FRT
11 - 15	RA
16 - 31	D

Item	Description
lfs	<i>FRT, D(RA)</i>

Description

The **lfs** 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

Item	Description
<i>FRT</i>	Specifies target floating-point register where data is stored.
<i>D</i>	16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	Specifies source general-purpose register for EA calculation.

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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other

operations.

“Floating-point load and store instructions” on page 29

Ifsu (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

Bits	Value
0 - 5	49
6 - 10	FRT
11 - 15	RA
16 - 31	D

Item	Description
Ifsu	<i>FRT</i> , <i>D</i> (<i>RA</i>)

Description

The **Ifsu** 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 **Ifsu** instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

Item	Description
<i>FRT</i>	Specifies target floating-point register where data is stored.
<i>D</i>	16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

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

Bits	Value
0 - 5	31
6 - 10	FRT
11 - 15	RA
16 - 20	RB
21 - 30	567
31	/

Item	Description
Ifsux	<i>FRT, RA, RB</i>

Description

The **Ifsux** 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 **Ifsux** instruction has one syntax form and does not affect the Floating-Point Status Control Register.

Parameters

Item	Description
<i>FRT</i>	Specifies target floating-point register where data is stored.
<i>RA</i>	Specifies source general-purpose register for EA calculation and possible address update.
<i>RB</i>	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:

```
.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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

lfsx (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

Bits	Value
0 - 5	31
6 - 10	<i>FRT</i>
11 - 15	<i>RA</i>
16 - 20	<i>RB</i>
21 - 30	535
31	/

Item	Description
<i>lfsx</i>	<i>FRT</i> , <i>RA</i> , <i>RB</i>

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

Item	Description

<i>FRT</i>	Specifies target floating-point register where data is stored.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

lha (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

Bits	Value
0 - 5	42
6 - 10	RT
11 - 15	RA
16 - 31	D

Item	Description

lha	<i>RT</i> , <i>D</i> (<i>RA</i>)

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

Item	Description

<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>D</i>	16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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]
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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

lhau (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

Bits	Value
0 - 5	43
6 - 10	<i>RT</i>
11 - 15	<i>RA</i>
16 - 31	<i>D</i>

Item	Description
lhau	<i>RT</i> , <i>D</i> (<i>RA</i>)

Description

The **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 **lhau** instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>D</i>	16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

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

Bits	Value
0 - 5	31
6 - 10	<i>RT</i>
11 - 15	<i>RA</i>
16 - 20	<i>RB</i>
21 - 30	375
31	/

Item	Description
lhax	<i>RT, RA, RB</i>

Description

The **lhax** 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 **lhax** instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies first source general-purpose register for EA calculation and possible address update.
<i>RB</i>	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]
lhax 6,5,4
# GPR 6 now contains 0xffff ffff.
# GPR 5 now contains the storage address.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

lhax (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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21 - 30	343
31	/

Item	Description
lhax	<i>RT, RA, RB</i>

Description

The **lhax** 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 **lhax** instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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]
lhax 6,5,4
# GPR 6 now contains 0x0000 0001.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

lhbrx (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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21 - 30	790
31	/

Item	Description
lhbrx	<i>RT, RA, RB</i>

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

lhz (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

Bits	Value
0 - 5	40
6 - 10	RT
11 - 15	RA
16 - 31	D

Item	Description
lhz	RT, D(RA)

Description

The **lhz** 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 **lhz** instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>D</i>	16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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:

```
.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 concepts:

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

lhzu (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

Bits	Value
0 - 5	41
6 - 10	RT
11 - 15	RA
16 - 31	D

Item	Description
lhzu	<i>RT</i> , <i>D</i> (<i>RA</i>)

Description

The **lhzu** 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 **lhzu** instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>D</i>	16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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:

```
.csect data[rw]
.short 0xffff
# Assume GPR 4 contains the address of csect data[rw].
.csect text[pr]
lhzu 6,0(4)
# GPR 6 now contains 0x0000 ffff.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

lhzux (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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21 - 30	331
31	/

Item	Description
lhzux	RT, RA, RB

Description

The **lhzux** 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 **lhzux** instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for EA calculation and possible address update.
<i>RB</i>	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:

```
.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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

lhzx (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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21 - 30	279
31	/

Item	Description
lhzx	RT, RA, RB

Description

The **lhzx** 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 **lhzx** instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

Item	Description
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:

```
.csect data[rw]
.short 0xffff
.csect text[pr]
```

```
# Assume GPR 5 contains the address of csect data[rw].
# Assume 0xffff is the halfword located at displacement 0.
# Assume GPR 4 contains 0x0000 0000.
lhcx 6,5,4
# GPR 6 now contains 0x0000 ffff.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

lmw or lm (Load Multiple Word) instruction

Purpose

Loads consecutive words at a specified location into more than one general-purpose register.

Syntax

Bits	Value
0 - 5	46
6 - 10	RT
11 - 15	RA
16 - 31	D

PowerPC®

lmw *RT, D(RA)*

POWER® family

lm *RT, D(RA)*

Description

The **lmw** and **lm** 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 **lmw**:

- 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 **lm**, 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

Item	Description
<i>RT</i>	Specifies starting target general-purpose register for operation.
<i>D</i>	Specifies a 16-bit signed two's complement integer sign extended to 32 bits for EA calculation
<i>RA</i>	Specifies source general-purpose register for EA calculation.

Examples

The following code loads data into GPR 29 and GPR 31:

```
.csect data[rw]
.long 0x8971
.long -1
.long 0x7ffe c100
# Assume GPR 30 contains the address of csect data[rw].
.csect text[pr]
lmw 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

lq (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

Bits	Value
0 - 5	56
6 - 10	RS
11 - 15	RA
16 - 27	DQ
28 - 31	0

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*.

DQ is a 12-bit, signed two's complement number, which is sign extended to 64 bits and then multiplied by 16 to provide a displacement *Disp*. If GPR *RA* is not 0, the EA is the sum of the contents of GPR *RA* and *Disp*. If GPR *RA* is 0, then the EA is *Disp*.

Parameters

Item Description

<i>RT</i>	Specifies target general-purpose register where result of operation is stored. If <i>RT</i> is odd, the instruction form is invalid.
<i>Disp</i>	Specifies a 16-bit signed number that is a multiple of 16. The assembler divides this number by 16 when generating the instruction.
<i>RA</i>	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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21 - 30	277
31	Rc

Item	Description
POWER family	POWER family
lscbx	<i>RT, RA, RB</i>
lscbx.	<i>RT, RA, RB</i>

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.

Item	Description	Description	Description	Description
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
lscbx	None	XER(25-31) = # of bytes loaded	0	None
lscbx.	None	XER(25-31) = # of bytes loaded	1	LT,GT,EQ,SO

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:

LT, GT, EQ, SO = b'00' || match || 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

Item	Description
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 0x6fXX 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 0xFFFF XXXX.
# GPR 8 now contains 0xFFFF XXXX.
# XER25-31 = 2.
# CRF 0 now contains 0x2.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point string instructions” on page 23

The Fixed-Point String instructions allow the movement of data from storage to registers or from registers to storage without concern for alignment.

lswi or lsi (Load String Word Immediate) instruction

Purpose

Loads consecutive bytes in storage from a specified location in memory into consecutive general-purpose registers.

Syntax

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	NB
21 - 30	597
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 $\text{ceiling}(N/4)$, 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

Item	Description
<i>RT</i>	Specifies starting general-purpose register of stored data.
<i>RA</i>	Specifies general-purpose register for EA calculation.
<i>NB</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point string instructions” on page 23

The Fixed-Point String instructions allow the movement of data from storage to registers or from registers to storage without concern for alignment.

lswx or lsx (Load String Word Indexed) instruction

Purpose

Loads consecutive bytes in storage from a specified location in memory into consecutive general-purpose registers.

Syntax

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21 - 30	533
31	/

PowerPC®

lswx *RT, RA, RB*

POWER® family
lsw *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

Item	Description
<i>RT</i>	Specifies starting general-purpose register of stored data.
<i>RA</i>	Specifies general-purpose register for EA calculation.
<i>RB</i>	Specifies general-purpose register for EA calculation.

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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point string instructions” on page 23

The Fixed-Point String instructions allow the movement of data from storage to registers or from registers to storage without concern for alignment.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

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

Bits	Value
0 - 5	58
6 - 10	RT
11 - 15	RA
16 - 29	DS
30 - 31	0b10

PowerPC 64

Iwa *RT, Disp (RA)*

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.

DS is a 14-bit, signed two's complement number, which is sign-extended to 64 bits, and then multiplied by 4 to provide a displacement *Disp*. If GPR *RA* is not 0, the EA is the sum of the contents of GPR *RA* and *Disp*. If GPR *RA* is 0, then the EA is *Disp*.

Parameters

Item	Description

<i>RT</i>	Specifies target general-purpose register where result of the operation is stored.

<i>Disp</i>	Specifies a 16-bit signed number that is a multiple of 4. The assembler divides this number by 4 when generating the instruction.

<i>RA</i>	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.

lwarx (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 **lwarx** instruction is supported only in the PowerPC® architecture.

Syntax

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21- 30	20
31	/

PowerPC®

lwarx *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

Item Description

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 concepts:

“stwcx. (Store Word Conditional Indexed) instruction” on page 480

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

Iwaux (Load Word Algebraic with Update Indexed) instruction

Purpose

Load a fullword of data from storage into the low-order 32b its of the specified general purpose register. Sign extend the data into the high-order 32 bits of the register. Update the address base.

Syntax

Bits	Value
0 - 5	31
6 - 10	D
11 - 15	A
16 - 20	B
21 - 30	373
31	0

POWER® family
lwaux 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 puspouse 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

Item	Description
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.

Syntax

Bits	Value
0 - 5	31
6 - 10	D
11 - 15	A
16 - 20	B
21 - 30	341
31	0

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 puspouse 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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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.

lwbrx or lbrx (Load Word Byte-Reverse Indexed) instruction

Purpose

Loads a byte-reversed word of data from a specified location in memory into a general-purpose register.

Syntax

Bits	Value
0 - 5	31
6 - 10	<i>RT</i>
11 - 15	<i>RA</i>
16 - 20	<i>RB</i>
21 - 30	534
31	/

PowerPC®

lwbrx *RT, RA, RB*

POWER® family

lbrx *RT, RA, RB*

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	Specifies source general-purpose register for EA calculation.

Examples

The following code loads a byte-reversed word from memory into GPR 6:

```
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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

lwz or l (Load Word and Zero) instruction

Purpose

Loads a word of data from a specified location in memory into a general-purpose register.

Syntax

Bits	Value
0 - 5	32
6 - 10	<i>RT</i>
11 - 15	<i>RA</i>
16 - 31	<i>D</i>

PowerPC®

lwz *RT, D(RA)*

POWER® family

l *RT, D(RA)*

Description

The **lwz** and **l** 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 **l** instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

Item	Description

<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>D</i>	Specifies a 16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

lwzu or lu (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

Bits	Value
0 - 5	33
6 - 10	<i>RT</i>
11 - 15	<i>RA</i>
16 - 31	<i>D</i>

PowerPC®

lwzu *RT, D(RA)*

POWER® family

lu *RT, D(RA)*

Description

The **lwzu** and **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) *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 **lwzu** and **lu** instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>D</i>	Specifies a 16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

lwzux 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

Bits	Value
0 - 5	31
6 - 10	<i>RT</i>
11 - 15	<i>RA</i>
16 - 20	<i>RB</i>
21 - 30	55
31	/

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for EA calculation and possible address update.
<i>RB</i>	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:

```
.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].
.csect text[pr]
lwzux 6,5,4
# GPR 6 now contains 0xffdd 75ce.
# GPR 5 now contains the storage address.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

lwzx or lx (Load Word and Zero Indexed) instruction

Purpose

Loads a word of data from a specified location in memory into a general-purpose register.

Syntax

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21 - 30	23
31	/

PowerPC®

lwzx *RT, RA, RB*

POWER® family

lx *RT, RA, RB*

Description

The **lwzx** and **lx** 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 **lwzx** and **lx** instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	Specifies source general-purpose register for EA calculation.

Examples

The following code loads a word from memory into GPR 6:

```
.csect data[rw]
.long 0xffdd 75ce
# Assume GPR 4 contains the displacement relative to
# csect data[rw].
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
lwzx 6,5,4
# GPR 6 now contains 0xffdd 75ce.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	29
31	Rc

POWER® family

maskg *RA, RS, RB*

maskg. *RA, RS, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
maskg	None	None	0	None
maskg.	None	None	1	LT,GT,EQ,SO

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

Item	Description
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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	541
31	Rc

POWER® family
maskir *RA, RS, RB*
maskir. *RA, RS, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
maskir	None	None	0	None
maskir.	None	None	1	LT, GT, EQ, SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26
The fixed-point rotate and shift instructions rotate the contents of a register.

mcrf (Move Condition Register Field) instruction

Purpose

Copies the contents of one condition register field into another.

Syntax

Bits	Value
0 - 5	19
6 - 8	BF
9 - 10	//
11 - 13	BFA
14 - 15	//
16 - 20	///
21 - 30	0
31	/

Item	Description
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

Item	Description
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:

```
# Assume Condition Register Field 3 holds b'0110'.  
mcrf 2,3  
# Condition Register Field 2 now holds b'0110'.
```

Related concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

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

Bits	Value
0 - 5	63
6 - 8	BF
9 - 10	//
11 - 13	BFA
14 - 15	//
16 - 20	///
21 - 30	64
31	/

Item	Description
mcrfs	<i>BF, BFA</i>

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.

BFA	FPSCR bits set
0	FX,OX
1	UX, ZX, XX, VXSNaN
2	VXISI, VXIDI, VXZDZ, VXIMZ
3	VXVC

Parameters

Item	Description
<i>BF</i>	Specifies target condition register field where result of operation is stored.
<i>BFA</i>	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:

```
# Assume FPSCR 4 contains b'0111'.
mcrfs 3,4
# Condition Register Field 3 contains b'0111'.
```

Related concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

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

Bits	Value
0 - 5	31
6 - 8	BF
9 - 10	//
11 - 15	///
16 - 20	///
21 - 30	512
31	/

Item	Description
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

Item	Description
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.

```
# 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 concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Fixed-point move to or from special-purpose registers instructions” on page 27

The instructions move the contents of one Special-Purpose Register (SPR) into another SPR or into a General-Purpose Register (GPR). These include both nonprivileged and privileged instructions.

mfcrr (Move from Condition Register) instruction

Purpose

Copies the contents of the Condition Register into a general-purpose register.

Syntax

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	///
16 - 20	///
21 - 30	19
31	Rc

Item	Description
mfcrr	RT

Description

The **mfcrr** instruction copies the contents of the Condition Register into target general-purpose register (GPR) *RT*.

The **mfcrr** instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters

Item	Description
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.  
mfcrr 6  
# GPR 6 now contains 0x4055 F605.
```

Related concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Fixed-point move to or from special-purpose registers instructions” on page 27

The instructions move the contents of one Special-Purpose Register (SPR) into another SPR or into a General-Purpose Register (GPR). These include both nonprivileged and privileged 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

Bits	Value
0 - 5	63
6 - 10	FRT
11 - 15	///
16 - 20	///
21 - 30	583
31	Rc

Item	Description
mffs	<i>FRT</i>
mffs.	<i>FRT</i>

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.

Item	Description		
Syntax Form	FPSCR bits	Record Bit (Rc)	Condition Register Field 1
mffs	None	0	None
mffs.	None	1	FX, FEX, VX, OX

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

Item	Description
<i>FRT</i>	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:

```
# Assume FPSCR contains 0x0000 0000.  
mffs 14  
# FPR 14 now contains 0xFFFF FFFF 0000 0000.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

“Functional differences for POWER[®] family and PowerPC[®] instructions” on page 144
The POWER[®] family and PowerPC[®] instructions that share the same op code on POWER[®] family and PowerPC[®] platforms, but differ in their functional definition.

mfmsr (Move from Machine State Register) instruction

Purpose

Copies the contents of the Machine State Register into a general-purpose register.

Syntax

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	///
16 - 20	///
21 - 30	83
31	/

Item	Description
mfmsr	RT

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

Item	Description
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 concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Fixed-point move to or from special-purpose registers instructions” on page 27

The instructions move the contents of one Special-Purpose Register (SPR) into another SPR or into a General-Purpose Register (GPR). These include both nonprivileged and privileged instructions.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

mfocrf (Move from One Condition Register Field) instruction

Purpose

Copies the contents of one Condition Register field into a general-purpose register.

Syntax

Bits	Value
0 - 5	31
6 - 10	RT
11	1
12 - 19	FXM
20	///
21 - 30	19
31	///

Item	Description
mfocrf	<i>RT, FXM</i>

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:

Bit	Description
12	CR 00-03 is copied into GPR <i>RS</i> 00-03.
13	CR 04-07 is copied into GPR <i>RS</i> 04-07.
14	CR 08-11 is copied into GPR <i>RS</i> 08-11.
15	CR 12-15 is copied into GPR <i>RS</i> 12-15.
16	CR 16-19 is copied into GPR <i>RS</i> 16-19.
17	CR 20-23 is copied into GPR <i>RS</i> 20-23.
18	CR 24-27 is copied into GPR <i>RS</i> 24-27.
19	CR 28-31 is copied into GPR <i>RS</i> 28-31.

The **mfocrf** instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>FXM</i>	Specifies field mask. Only one bit may be specified.

Examples

The following code copies the Condition Register field 3 into GPR 6:

```
# Assume the Condition Register contains 0x4055 F605.
# Field 3 (0x10 = b'0001 0000')
mfocrf 6, 0x10
# GPR 6 now contains 0x0005 0000.
```

Related concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Fixed-point move to or from special-purpose registers instructions” on page 27

The instructions move the contents of one Special-Purpose Register (SPR) into another SPR or into a General-Purpose Register (GPR). These include both nonprivileged and privileged instructions.

mfspr (Move from Special-Purpose Register) instruction

Purpose

Copies the contents of a special-purpose register into a general-purpose register.

Syntax

Bits	Value
0 - 5	31
6 - 10	RT
11 - 20	spr
21 - 30	339
31	Rc

Item	Description
mfspr	<i>RT</i> , <i>SPR</i>

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.

Item	Description	Description	Description
SPR Values	SPR Values	SPR Values	SPR Values
Decimal	$\text{spr}^{5:9} \text{spr}^{0:4}$	Register Name	Privileged
1	00000 00001	XER	No
8	00000 01000	LR	No
9	00000 01001	CTR	No
18	00000 10010	DSISR	Yes
19	00000 10011	DAR	Yes
22	00000 10110	DEC ²	Yes
25	00000 11001	SDR1	Yes
26	00000 11010	SRR0	Yes
27	00000 11011	SRR1	Yes
272	01000 10000	SPRG0	Yes
273	01000 10001	SPRG1	Yes
274	01000 10010	SPRG2	Yes
275	01000 10011	SPRG3	Yes
282	01000 11010	EAR	Yes
284	01000 11100	TBL	Yes
285	01000 11101	TBU	Yes
528	10000 10000	IBAT0U	Yes
529	10000 10001	IBAT0L	Yes
530	10000 10010	IBAT1U	Yes
531	10000 10011	IBAT1L	Yes
532	10000 10100	IBAT2U	Yes
533	10000 10101	IBAT2L	Yes
534	10000 10110	IBAT3U	Yes
535	10000 10111	IBAT3L	Yes
536	10000 11000	DBAT0U	Yes
537	10000 11001	DBAT0L	Yes
538	10000 11010	DBAT1U	Yes
539	10000 11011	DBAT1L	Yes
540	10000 11100	DBAT2U	Yes
541	10000 11101	DBAT2L	Yes
542	10000 11110	DBAT3U	Yes
543	10000 11111	DBAT3L	Yes
0	00000 00000	MQ ¹	No
4	00000 00100	RTCU ¹	No
5	00000 00101	RTCL ¹	No
6	00000 00110	DEC ²	No

¹Supported only in the POWER family architecture.

²In the PowerPC architecture moving from the DEC register is privileged and the SPR value is 22. In the POWER family architecture moving from the DEC register is not privileged and the SPR value is 6. For more information, see Fixed-Point Move to or from Special-Purpose Registers Instructions .

If the SPR field contains any value other than those listed in the SPR Values table, the instruction form is invalid.

The **mf spr** instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

Item	Description

<i>RT</i>	Specifies target general-purpose register where result of operation is stored.

<i>SPR</i>	Specifies source special-purpose register for operation.

Examples

The following code copies the contents of the Fixed-Point Exception Register into GPR 6:

```
mf spr 6,1
# GPR 6 now contains the bit settings of the Fixed
# Point Exception Register.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point move to or from special-purpose registers instructions” on page 27

The instructions move the contents of one Special-Purpose Register (SPR) into another SPR or into a General-Purpose Register (GPR). These include both nonprivileged and privileged instructions.

mf sr (Move from Segment Register) instruction

Purpose

Copies the contents of a segment register into a general-purpose register.

Syntax

Bits	Value
0 - 5	31
6 - 8	RT
11	/
12 - 14	SR
16 - 20	///
21 - 30	595
31	/

Item	Description

mf sr	<i>RT, SR</i>

Description

The **mf sr** instruction copies the contents of segment register (SR) into target general-purpose register (GPR) *RT*.

The **mf sr** 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

Item	Description
<i>RT</i>	Specifies the target general-purpose register where the result of the operation is stored.
<i>SR</i>	Specifies the source segment register for the operation.

Examples

The following code copies the contents of Segment Register 7 into GPR 6:

```
# 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 concepts:

“mfsri (Move from Segment Register Indirect) instruction”

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	627
31	Rc

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

Item	Description
<i>RS</i>	Specifies the target general-purpose register for operation.
<i>RA</i>	Specifies the source general-purpose register for SR calculation.
<i>RB</i>	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 concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“mfsrin (Move from 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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	///
16 - 20	RB
21 - 30	659
31	/

PowerPC®

mfsrin *RT, RB*

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

Item	Description

<i>RT</i>	Specifies the target general-purpose register for operation.

<i>RB</i>	Specifies the source general-purpose register for SR calculation.

Security

The **mfsrin** instruction is privileged.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“mfspr extended mnemonics for POWER® family” on page 132

mfspr Extended Mnemonics for POWER® family

“mfsrin (Move from Segment Register Indirect) instruction” on page 350

mtrcf (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

Bits	Value
0 - 5	31
6 - 10	RS
11	/
12 - 19	FXM
20	/
21 - 30	144
31	Rc

Item	Description
mtrcf	<i>FXM, RS</i>

See Extended Mnemonics of Condition Register Logical Instructions for more information.

Description

The **mtrcf** 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:

Bit	Description
12	CR 00-03 is updated with the contents of GPR RS 00-03.
13	CR 04-07 is updated with the contents of GPR RS 04-07.
14	CR 08-11 is updated with the contents of GPR RS 08-11.
15	CR 12-15 is updated with the contents of GPR RS 12-15.
16	CR 16-19 is updated with the contents of GPR RS 16-19.
17	CR 20-23 is updated with the contents of GPR RS 20-23.
18	CR 24-27 is updated with the contents of GPR RS 24-27.
19	CR 28-31 is updated with the contents of GPR RS 28-31.

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

Item	Description
<i>FXM</i>	Specifies field mask.
<i>RS</i>	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').
mtcrf 0x80,5
# Condition Register Field 0 now contains b'0111'.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Fixed-point move to or from special-purpose registers instructions” on page 27

The instructions move the contents of one Special-Purpose Register (SPR) into another SPR or into a General-Purpose Register (GPR). These include both nonprivileged and privileged instructions.

mtfsb0 (Move to FPSCR Bit 0) instruction

Purpose

Sets a specified Floating-Point Status and Control Register bit to 0.

Syntax

Bits	Value
0 - 5	63
6 - 10	BT
11 - 15	///
16 - 20	///
21 - 30	70
31	Rc

Item	Description
mtfsb0	<i>BT</i>
mtfsb0.	<i>BT</i>

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.

Item	Description		
Syntax Form	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 1
mtfsb0	None	0	None
mtfsb0.	None	1	FX, FEX, VX, OX

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

Item	Description
<i>BT</i>	Specifies Floating-Point Status and Control Register bit set by operation.

Examples

- 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.
```
- 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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

mtfsb1 (Move to FPSCR Bit 1) instruction

Purpose

Sets a specified Floating-Point Status and Control Register bit to 1.

Syntax

Bits	Value
0-5	63
6-10	BT
11-15	///
16-20	///
21-30	38
31	Rc

Item	Description
mtfsb1	<i>BT</i>
mtfsb1.	<i>BT</i>

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.

Item	Description		
Syntax Form	FPSCR Bits	Record Bit (Rc)	Condition Register Field 1
mtfsb1	None	0	None
mtfsb1.	None	1	FX, FEX, VX, OX

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

Item	Description
<i>BT</i>	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:

```
mtfsbl 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:

```
mtfsbl. 3
# Now bit 3 of the Floating-Point Status and Control
# Register is set to 1.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

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

Bits	Value
0 - 5	63
6	/
7 - 14	FLM
15	/
16 - 20	FRB
21 - 30	771
31	Rc

Item	Description
mtfsf	<i>FLM</i> , <i>FRB</i>
mtfsf.	<i>FLM</i> , <i>FRB</i>

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:

Bit	Description

- 7 FPSCR 00-03 is updated with the contents of *FRB* 32-35.
- 8 FPSCR 04-07 is updated with the contents of *FRB* 36-39.
- 9 FPSCR 08-11 is updated with the contents of *FRB* 40-43.
- 10 FPSCR 12-15 is updated with the contents of *FRB* 44-47.
- 11 FPSCR 16-19 is updated with the contents of *FRB* 48-51.
- 12 FPSCR 20-23 is updated with the contents of *FRB* 52-55.
- 13 FPSCR 24-27 is updated with the contents of *FRB* 56-59.
- 14 FPSCR 28-31 is updated with the contents of *FRB* 60-63.

The **mtfsf** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Item	Description		
Syntax Form	FPSCR Bits	Record Bit (Rc)	Condition Register Field 1
mtfsf	None	0	None
mtfsf.	None	1	FX, FEX, VX, OX

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

Item Description

FLM Specifies field mask.

FRB Specifies source floating-point register for operation.

Examples

- The following code copies the contents of floating-point register 5 bits 32-35 into Floating-Point Status and Control Register Field 0:

```
# 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'.
```

- 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:

```
# Assume bits 32-63 of FPR 5
# contains 0x2320 0000.
mtfsf. 0xE0,5
# Floating-Point Status and Control Register Fields 0-2
# now contain b'0010 0011 0010'.
# Condition Register Field 1 now contains 0x2.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the

instruction.

mtfsfi (Move to FPSCR Field Immediate) instruction

Purpose

Copies an immediate value into a specified Floating-Point Status and Control Register field.

Syntax

Bits	Value
0 - 5	63
6 - 8	BF
9 - 10	//
11 - 15	///
16 - 19	U
20	/
21 - 30	134
31	Rc

Item	Description
mtfsfi	<i>BF, I</i>
mtfsfi.	<i>BF, I</i>

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.

Item	Description		
Syntax Form	FPSCR Bits	Record Bit (Rc)	Condition Register Field 1
mtfsfi	None	0	None
mtfsfi.	None	1	FX, FEX, VX, OX

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

Item	Description
<i>BF</i>	Specifies target Floating-Point Status and Control Register field for operation.
<i>I</i>	Specifies source immediate value for operation.

Examples

- 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'.
```
- 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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Interpreting the contents of a floating-point register” on page 29

There are thirty-two 64-bit floating-point registers. The floating-point register is used to execute the instruction.

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

Bits	Value
0 - 5	31
6 - 10	RT
11	/
12 - 19	FXM
20	/
21 - 30	144
31	/

Item	Description
mtocrf	<i>FXM</i> , <i>RS</i>

Description

The **mtocrf** 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:

Bit	Description
12	CR 00-03 is updated with the contents of GPR <i>RS</i> 00-03.
13	CR 04-07 is updated with the contents of GPR <i>RS</i> 04-07.
14	CR 08-11 is updated with the contents of GPR <i>RS</i> 08-11.
15	CR 12-15 is updated with the contents of GPR <i>RS</i> 12-15.
16	CR 16-19 is updated with the contents of GPR <i>RS</i> 16-19.
17	CR 20-23 is updated with the contents of GPR <i>RS</i> 20-23.
18	CR 24-27 is updated with the contents of GPR <i>RS</i> 24-27.
19	CR 28-31 is updated with the contents of GPR <i>RS</i> 28-31.

The **mtocrf** instruction has one syntax form and does not affect the Fixed-Point Exception Register.

Parameters

Item	Description
<i>FXM</i>	Specifies field mask.
<i>RS</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point move to or from special-purpose registers instructions” on page 27

The instructions move the contents of one Special-Purpose Register (SPR) into another SPR or into a General-Purpose Register (GPR). These include both nonprivileged and privileged instructions.

mtspr (Move to Special-Purpose Register) instruction

Purpose

Copies the contents of a general-purpose register into a special-purpose register.

Syntax

Bits	Value
0 - 5	31
6 - 10	<i>RS</i>
11 - 20	<i>spr</i>
21 - 30	467
31	<i>Rc</i>

Item	Description
mtspr	<i>SPR, RS</i>

Note: The special-purpose register is a split field.

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.

Item	Description	Description	Description
SPR Values	SPR Values	SPR Values	SPR Values
Decimal	$\text{spr}^{5:9} \text{spr}^{0:4}$	Register Name	Privileged
1	00000 00001	XER	No
8	00000 01000	LR	No
9	00000 01001	CTR	No
18	00000 10010	DSISR	Yes
19	00000 10011	DAR	Yes
22	00000 10110	DEC	Yes ¹
25	00000 11001	SDR1	Yes
26	00000 11010	SRR0	Yes
27	00000 11011	SRR1	Yes
272	01000 10000	SPRG0	Yes
273	01000 10001	SPRG1	Yes
274	01000 10010	SPRG2	Yes
275	01000 10011	SPRG3	Yes
282	01000 11010	EAR	Yes
284	01000 11100	TBL	Yes
285	01000 11101	TBU	Yes
528	10000 10000	IBAT0U	Yes
529	10000 10001	IBAT0L	Yes
530	10000 10010	IBAT1U	Yes
531	10000 10011	IBAT1L	Yes
532	10000 10100	IBAT2U	Yes
533	10000 10101	IBAT2L	Yes
534	10000 10110	IBAT3U	Yes
535	10000 10111	IBAT3L	Yes
536	10000 11000	DBAT0U	Yes
537	10000 11001	DBAT0L	Yes
538	10000 11010	DBAT1U	Yes
539	10000 11011	DBAT1L	Yes
540	10000 11100	DBAT2U	Yes
541	10000 11101	DBAT2L	Yes
542	10000 11110	DBAT3U	Yes

Item	Description	Description	Description
543	10000 11111	DBAT3L	Yes
0	00000 00000	MQ ²	No
20	00000 10100	RTCU ²	Yes
21	00000 10101	RTCL ²	Yes

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

Item Description

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:

```
# Assume GPR 5 holds 0x1000 00FF.
mtspr 8,5
# The Link Register now holds 0x1000 00FF.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point move to or from special-purpose registers instructions” on page 27

The instructions move the contents of one Special-Purpose Register (SPR) into another SPR or into a General-Purpose Register (GPR). These include both nonprivileged and privileged 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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21	OE
22 - 30	107
31	Rc

POWER® family

mul	<i>RT, RA, RB</i>
mul.	<i>RT, RA, RB</i>
mulo	<i>RT, RA, RB</i>
mulo.	<i>RT, RA, RB</i>

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
mul	0	None	0	None
mul.	0	None	1	LT,GT,EQ,SO
mulo	1	SO,OV	0	None
mulo.	1	SO,OV	1	LT,GT,EQ,SO

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where the result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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:

```
# 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:

```
# 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.
```

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.
# Assume XER = 0.
mulo 6,4,10
# MQ Register now contains 0x1E30 0000.
# GPR 6 now contains 0xFFFF DD80.
# 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 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.
# Condition Register Field 0 now contains 0x5.
# XER now contains 0xc000 0000.
```

Related concepts:

“mul (Multiply) instruction” on page 361

“mulhw (Multiply High Word) instruction” on page 365

“mulhwu (Multiply High Word Unsigned) instruction” on page 367

“mulli or muli (Multiply Low Immediate) instruction” on page 369

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

“Using milicode routines” on page 108

The milicode routines contain machine-dependent and performance-critical functions.

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

Bits	Value
0-5	31
6-10	D
11-15	A
16-20	B
21	0
22-30	73
31	Rc

POWER® family

mulhd *RT, RA, RB* (Rc=0)

mulhd. *RT, RA, RB* (Rc=1)

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

Item	Description
<i>RT</i>	Specifies target general-purpose register for the result of the computation.
<i>RA</i>	Specifies source general-purpose register for an operand.
<i>RB</i>	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

Bits	Value
0 - 5	31
6 - 10	D
11 - 15	A
16 - 20	B
21	0
22 - 30	9
31	Rc

POWER® family

mulhdu *RT, RA, RB* (Rc=0)

mulhdu. *RT, RA, RB* (Rc=1)

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

Item	Description

<i>RT</i>	Specifies target general-purpose register for the result of the computation.

<i>RA</i>	Specifies source general-purpose register for the multiplicand.

<i>RB</i>	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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21	/
22 - 30	75
31	Rc

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.

Item	Description	
Syntax Form	Record Bit (Rc)	Condition Register Field 0
mulhw	0	None
mulhw.	1	LT,GT,EQ,SO

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

Item Description

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 concepts:

“mul (Multiply) instruction” on page 361

“mulhwu (Multiply High Word Unsigned) instruction” on page 367

“mulli or muli (Multiply Low Immediate) instruction” on page 369

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21	/
22 - 30	11
31	Rc

PowerPC®

mulhwu *RT, RA, RB*

mulhwu. *RT, RA, RB*

Description

The **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.

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 **mulhwu** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Item	Description	
Syntax Form	Record Bit (Rc)	Condition Register Field 0
mulhwu	0	None
mulhwu.	1	LT,GT,EQ,SO

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

Item	Description
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 contains 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 0x0000 4500.
# Assume GPR 10 contains 0x8000 7000.
# Assume XER(S0) = 0.
mulhwu. 6,4,10
# GPR 6 now contains 0x0000 2280.
# Condition Register Field 0 now contains 0x4.
```

Related concepts:

“mul (Multiply) instruction” on page 361

“mulhw (Multiply High Word) instruction” on page 365

“mulhwu (Multiply High Word Unsigned) instruction” on page 367

“mulli or muli (Multiply Low Immediate) instruction” on page 369

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	31
6 - 10	D
11 - 15	A
16 - 20	B
21	OE
22 - 30	233
31	Rc

POWER® family

mulld	<i>RT, RA, RB</i> (OE=0 Rc=0)
mulld.	<i>RT, RA, RB</i> (OE=0 Rc=1)
mulldo	<i>RT, RA, RB</i> (OE=1 Rc=0)
mulldo.	<i>RT, RA, RB</i> (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 **mulldo** 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

Item	Description
<i>RT</i>	Specifies target general-purpose register for the result of the computation.
<i>RA</i>	Specifies source general-purpose register for an operand.
<i>RB</i>	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

Bits	Value
0 - 5	07
6 - 10	RT
11 - 15	RA
16 - 31	SI

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>SI</i>	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:

```
# Assume GPR 4 holds 0x0000 3000.
mulli 6,4,10
# GPR 6 now holds 0x0001 E000.
```

Related concepts:

“mul (Multiply) instruction” on page 361

“mulhw (Multiply High Word) instruction” on page 365

“mulhwu (Multiply High Word Unsigned) instruction” on page 367

“mulli or muli (Multiply Low Immediate) instruction” on page 369

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

mullw or muls (Multiply Low Word) instruction

Purpose

Computes the least significant 32 bits of the 64-bit product of two 32-bit integers.

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21	OE
22 - 30	235
31	Rc

PowerPC®

mullw *RT, RA, RB*
mullw. *RT, RA, RB*
mullwo *RT, RA, RB*
mullwo. *RT, RA, RB*

POWER® family

muls *RT, RA, RB*
muls. *RT, RA, RB*
mulso *RT, RA, RB*
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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
mullw	0	None	0	None
mullw.	0	None	1	LT,GT,EQ
mullwo	1	SO,OV	0	None
mullwo.	1	SO,OV	1	LT,GT,EQ
muls	0	None	0	None
muls.	0	None	1	LT,GT,EQ
mulso	1	SO,OV	0	None
mulso.	1	SO,OV	1	LT,GT,EQ

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

Item	Description

<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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:

```
# 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:

```
# Assume GPR 4 holds 0x0000 4500.  
# Assume GPR 10 holds 0x0000 7000.  
# Assume XER(S0) = 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:

```
# 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:

```
# 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 concepts:

“mul (Multiply) instruction” on page 361

“mulhw (Multiply High Word) instruction” on page 365

“mulhwu (Multiply High Word Unsigned) instruction” on page 367

“mulli or muli (Multiply Low Immediate) instruction” on page 369

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	///
21	OE
22 - 30	488
31	Rc

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
nabs	0	None	0	None
nabs.	0	None	1	LT,GT,EQ,SO
nabso	1	SO,OV	0	None
nabso.	1	SO,OV	1	LT,GT,EQ,SO

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

Item	Description
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.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	476
31	Rc

Item	Description
nand	<i>RA, RS, RB</i>
nand.	<i>RA, RS, RB</i>

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
nand	None	None	0	None
nand.	None	None	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>RB</i>	Specifies source general-purpose register for operation.

Examples

- 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.
```
- 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	///
21	OE
22 - 30	104
31	Rc

Item	Description
neg	<i>RT, RA</i>
neg.	<i>RT, RA</i>
nego	<i>RT, RA</i>
nego.	<i>RT, RA</i>

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
neg	0	None	0	None
neg.	0	None	1	LT,GT,EQ,SO
nego	1	SO,OV	0	None
nego.	1	SO,OV	1	LT,GT,EQ,SO

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.

Examples

- 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.
```
- 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.
```
- 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.
```
- 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	124
31	Rc

Item	Description
nor	<i>RA, RS, RB</i>
nor.	<i>RA, RS, RB</i>

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
nor	None	None	0	None
nor.	None	None	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>RB</i>	Specifies source general-purpose register for operation.

Examples

- The following code NORs the contents of GPR 4 and GPR 7 and stores the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 6 contains 0x789A 789B.
nor 6,4,7
# GPR 7 now contains 0x0765 8764.
```
- 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 0xB004 3000.
# Assume GPR 7 contains 0x789A 789B.
nor. 6,4,7
# GPR 6 now contains 0x0761 8764.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

or (OR) instruction

Purpose

Logically ORs the contents of two general-purpose registers and stores the result in another general-purpose register.

Syntax

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	444
31	Rc

Item	Description
or	<i>RA, RS, RB</i>
or.	<i>RA, RS, RB</i>

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
or	None	None	0	None
or.	None	None	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	412
31	Rc

Item	Description
orc	<i>RA, RS, RB</i>
orc.	<i>RA, RS, RB</i>

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
orc	None	None	0	None
orc.	None	None	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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 0xB004 3000.
# Assume GPR 7 contains 0x789A 789B, whose
# complement is 0x8765 8764.
orc. 6,4,7
# GPR 6 now contains 0xB765 B764.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

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

Bits	Value
0 - 5	24
6 - 10	RS
11 - 15	RA
16 - 31	UI

PowerPC®

ori *RA, RS, UI*

POWER® family
oril *RA, RS, UI*

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>UI</i>	Specifies a16-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:

```
# Assume GPR 4 contains 0x9000 3000.  
ori 6,4,0x0079  
# GPR 6 now contains 0x9000 3079.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

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

Bits	Value
0 - 5	25
6 - 10	RS
11 - 15	RA
16 - 31	UI

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>UI</i>	Specifies a16-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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	///
21 - 30	122
31	/

POWER5™

Item	Description
popcntbd	<i>RS, RA</i>

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

Item	Description
<i>RS</i>	Specifies source general-purpose register.
<i>RA</i>	Specifies destination general-purpose register.

Related concepts:

“cntlzw or cntlz (Count Leading Zeros Word) instruction” on page 196

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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21 - 30	818
31	Rc

POWER® family

rac *RT, RA, RB*
rac. *RT, RA, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
rac	None	None	0	None
rac.	None	None	1	EQ,SO

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

Item Description

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 concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

rfi (Return from Interrupt) instruction

Purpose

Reinitializes the Machine State Register and continues processing after an interrupt.

Syntax

Bits	Value
0 - 5	19
6 - 10	///
11 - 15	///
16 - 20	///
21 - 30	50
31	/

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 concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

rfid (Return from Interrupt Double Word) instruction

Purpose

Reinitializes the Machine State Register and continues processing after an interrupt.

Syntax

Bits	Value
0 - 5	19
6 - 10	00000
11 - 15	00000
16 - 20	00000
21 - 30	18
31	0

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

Bits	Value
0 - 5	19
6 - 10	///
11 - 15	///
16 - 20	///
21 - 30	82
31	LK

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 concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“System call instruction” on page 20

The PowerPC[®] system call instructions generate an interrupt or the system to perform a service.

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

Bits	Value
0 - 5	30
6 - 10	S
11 - 15	A
16 - 20	B
21 - 26	mb
27 - 30	8
31	Rc

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 **rdcl** 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

Item	Description
<i>RA</i>	Specifies the target general purpose register for the result of the instruction.
<i>RS</i>	Specifies the source general purpose register containing the operand.
<i>RB</i>	Specifies the source general purpose register containing the shift value.
<i>MB</i>	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

Bits	Value
0 - 5	30
6 - 10	S
11 - 15	A
16 - 20	sh
21 - 26	mb
27 - 29	0
30	sh
31	Rc

PowerPC64

rldicl *rA, rS, rB, MB* (Rc=0)
rldicl. *rA, rS, rB, MB* (Rc=1)

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 **rldicl** 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

Item	Description
<i>rA</i>	***DESCRIPTION***
<i>rS</i>	***DESCRIPTION***
<i>rB</i>	***DESCRIPTION***
<i>MB</i>	***DESCRIPTION***

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

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

Bits	Value
0 - 5	30
6 - 10	S
11 - 15	A
16 - 20	B
21 - 26	me
27 - 30	9
31	Rc

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

Item Description

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

Bits	Value
0 - 5	30
6 - 10	S
11 - 15	A
16 - 20	sh
21 - 26	mb
27 - 29	2
30	sh
31	Rc

POWER® family

rldicl *RA, RS, SH, MB* (Rc=0)

rldicl. *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 *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 **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

Item	Description
<i>RA</i>	Specifies the target general purpose register for the result of the instruction.
<i>RS</i>	Specifies the source general purpose register containing the operand.
<i>SH</i>	Specifies the (immediate) shift value for the operation.
<i>MB</i>	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.

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

Bits	Value
0 - 5	30
6 - 10	S
11 - 15	A
16 - 20	sh
21 - 26	mb
27 - 29	0
30	sh
31	Rc

POWER® family

rldicl *RA, RS, SH, MB* (Rc=0)

rldicl. *RA, RS, SH, MB* (Rc=1)

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 **rldicl** 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

Item	Description
<i>RA</i>	Specifies the target general purpose register for the result of the instruction.
<i>RS</i>	Specifies the source general purpose register containing the operand.
<i>SH</i>	Specifies the (immediate) shift value for the operation.
<i>MB</i>	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) instruction

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

Bits	Value
0 - 5	30
6 - 10	S
11 - 15	A
16 - 20	sh
21 - 26	me
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

Item	Description
<i>RA</i>	Specifies the target general purpose register for the result of the instruction.
<i>RS</i>	Specifies the source general purpose register containing the operand.
<i>SH</i>	Specifies the (immediate) shift value for the operation.
<i>ME</i>	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

Bits	Value
0 - 5	30
6 - 10	S
11 - 15	A
16 - 20	sh
21 - 26	mb
27 - 29	3
30	sh
31	Rc

POWER® family

rldimi *RA, RS, SH, MB* (Rc=0)
rldimi. *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 *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

Item	Description
<i>RA</i>	Specifies the target general purpose register for the result of the instruction.
<i>RS</i>	Specifies the source general purpose register containing the operand.
<i>SH</i>	Specifies the (immediate) shift value for the operation.
<i>MB</i>	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

Bits	Value
0 - 5	22
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 25	MB
26 - 30	ME
31	Rc

POWER® family

rlmi	<i>RA, RS, RB, MB, ME</i>
rlmi.	<i>RA, RS, RB, MB, ME</i>
rlmi	<i>RA, RS, RB, BM</i>
rlmi.	<i>RA, RS, RB, BM</i>

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
rlmi	None	None	0	None
rlmi.	None	None	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>RB</i>	Specifies general-purpose register that contains number of bits for rotation of data.
<i>MB</i>	Specifies begin value of mask for operation.
<i>ME</i>	Specifies end value of mask for operation.
<i>BM</i>	Specifies value of 32-bit mask.

Examples

- 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
# rlm 6,4,5,0xFFFFF0
# will produce the same result.
```
- 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
# rlm. 6,4,5,0xFFFFF0
# will produce the same result.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	20
6 - 10	RS
11 - 15	RA
16 - 20	SH
21 - 25	ME
26 - 30	MB
31	Rc

PowerPC®

rlwimi *RA, RS, SH, MB, ME*
rlwimi. *RA, RS, SH, MB, ME*
rlwimi *RA, RS, SH, BM*
rlwimi. *RA, RS, SH, BM*

POWER® family

rlimi *RA, RS, SH, MB, ME*
rlimi. *RA, RS, SH, MB, ME*
rlimi *RA, RS, SH, BM*
rlimi. *RA, RS, SH, BM*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
rlwimi	None	None	0	None
rlwimi.	None	None	1	LT,GT,EQ,SO
rlimi	None	None	0	None
rlimi.	None	None	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>SH</i>	Specifies shift value for operation.
<i>MB</i>	Specifies begin value of mask for operation.
<i>ME</i>	Specifies end value of mask for operation.
<i>BM</i>	Specifies value of 32-bit mask.

Examples

- 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 6,4,2,0xFFFFF0FC
# will produce the same result.
```
- 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. 6,4,2,0xFFFFF0E0
# will produce the same result.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	21
6 - 10	RS
11 - 15	RA
16 - 20	SH
21 - 25	MB
26 - 30	ME
31	Rc

PowerPC®

rlwinm *RA, RS, SH, MB, ME*
rlwinm. *RA, RS, SH, MB, ME*
rlwinm *RA, RS, SH, BM*
rlwinm. *RA, RS, SH, BM*

POWER® family

rlinm *RA, RS, SH, MB, ME*
rlinm. *RA, RS, SH, MB, ME*
rlinm *RA, RS, SH, BM*
rlinm. *RA, RS, SH, BM*

See Extended Mnemonics of Fixed-Point Rotate and Shift Instructions for more information.

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
rlwinm	None	None	0	None
rlwinm.	None	None	1	LT,GT,EQ,SO
rlinm	None	None	0	None
rlinm.	None	None	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>SH</i>	Specifies shift value for operation.
<i>MB</i>	Specifies begin value of mask for operation.
<i>ME</i>	Specifies end value of mask for operation.
<i>BM</i>	Specifies value of 32-bit mask.

Examples

- 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.
rlwinm 6,4,2,0,0x1D
# GPR 6 now contains 0x4000 C000.
# Under the same conditions
# rlwinm 6,4,2,0xFFFFFFC
# will produce the same result.
```
- 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.
rlwinm. 6,4,2,0,0x1D
# GPR 6 now contains 0xC010 C000.
# CRF 0 now contains 0x8.
# Under the same conditions
# rlwinm. 6,4,2,0xFFFFFFC
# will produce the same result.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

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

Bits	Value
0 - 5	23
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 25	MB
26 - 30	ME
31	Rc

PowerPC®

rlwnm *RA, RS, RB, MB, ME*
rlwnm. *RA, RS, RB, MB, ME*
rlwnm *RA, RS, SH, BM*
rlwnm. *RA, RS, SH, BM*

POWER® family

rlnm *RA, RS, RB, MB, ME*
rlnm. *RA, RS, RB, MB, ME*
rlnm *RA, RS, SH, BM*
rlnm. *RA, RS, SH, BM*

See Extended Mnemonics of Fixed-Point Rotate and Shift Instructions for more information.

Description

The **rlwnm** and **rlnm** instructions rotate 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*, 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 **rlwnm** and **rlnm** 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 **rlwnm** and **rlnm** instructions each have two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
rlwnm	None	None	0	None
rlwnm.	None	None	1	LT,GT,EQ,SO
rlnm	None	None	0	None
rlnm.	None	None	1	LT,GT,EQ,SO

The syntax forms of the **rlwnm** and **rlnm** 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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>RB</i>	Specifies general-purpose register that contains number of bits for rotation of data.
<i>MB</i>	Specifies begin value of mask for operation.
<i>ME</i>	Specifies end value of mask for operation.
<i>SH</i>	Specifies shift value for operation.
<i>BM</i>	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 0x4000 C000.
# Under the same conditions
#   rlwnm 6,4,5,0xFFFFF0
# 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. 6,4,5,0xFFFFF0
# will produce the same result.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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 **rrib** instruction is supported only in the POWER® family architecture.

Syntax

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	537
31	Rc

POWER® family

rrib *RA, RS, RB*

rrib. *RA, RS, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
rrib	None	None	0	None
rrib.	None	None	1	LT,GT,EQ,SO

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

Item	Description
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 0xB004 3000.
# Assume GPR 6 contains 0x0000 0004.
# Assume GPR 4 contains 0x0000 0000.
rrib. 4,5,6
# GPR 4 now contains 0x0800 0000.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

sc (System Call) instruction

Purpose

Calls the system to provide a service.

Note: The **sc** instruction is supported only in the PowerPC® architecture.

Syntax

Bits	Value
0 - 5	17
6 - 10	///
11 - 15	///
16 - 19	///
20 - 26	LEV
27 - 29	///
30	1
31	/

PowerPC®

Item	Description
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 496.

Parameters

Item	Description
LEV	Must be 0 or 1.

Related concepts:

“svc (Supervisor Call) instruction” on page 496

“System call instruction” on page 20

The PowerPC® system call instructions generate an interrupt or the system to perform a service.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

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

Bits	Value
0 - 5	17
6 - 10	///
11 - 15	///
16 - 19	///
20 -26	LEV
27 - 29	///
30	0
31	1

PowerPC®

Item	Description
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 “scv (System Call Vectored) instruction” on page 406.

Parameters

Item	Description
LEV	Must be 0 or 1.

Related concepts:

“svc (Supervisor Call) instruction” on page 496

“System call instruction” on page 20

The PowerPC® system call instructions generate an interrupt or the system to perform a service.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

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

Bits	Value
0 - 5	12
6 - 10	RT
11 - 15	RA
16 - 31	SI

Item	Description
si	<i>RT, RA, SINT</i>

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

Item	Description
<i>RT</i>	Specifies target general-purpose register for operation.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>SINT</i>	Specifies 16-bit signed integer for operation.
<i>SI</i>	Specifies the negative of the <i>SINT</i> 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:

```
# 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 concepts:

“addic or ai (Add Immediate Carrying) instruction” on page 163

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	13
6 - 10	RT
11 - 15	RA
16 - 31	SI

Item	Description
si.	<i>RT, RA, SINT</i>

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

Item	Description
<i>RT</i>	Specifies target general-purpose register for operation.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>SINT</i>	Specifies 16-bit signed integer for operation.
<i>SI</i>	Specifies the negative of the <i>SINT</i> 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:

```
# 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 concepts:

“addic or ai (Add Immediate Carrying) instruction” on page 163

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	31
6 - 10	S
11 - 15	A
16 -20	B
21 - 30	27
31	Rc

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

Item	Description

<i>RA</i>	Specifies target general-purpose register for the result of the operation.

<i>RS</i>	Specifies source general-purpose register containing the operand for the shift operation.

<i>RB</i>	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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	153
31	Rc

POWER® family

sle *RA, RS, RB*

sle. *RA, RS, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
sle	None	None	0	None
sle.	None	None	1	LT,GT,EQ,SO

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

Item Description

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 0xB004 3000.
# Assume GPR 5 contains 0x0000 0004.
sle. 6,4,5
# GPR 6 now contains 0x0043 0000.
# The MQ Register now contains 0x0043 000B.
# Condition Register Field 0 now contains 0x4.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	217
31	Rc

POWER® family

sleq *RA, RS, RB*

sleq. *RA, RS, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
sleq	None	None	0	None
sleq.	None	None	1	LT,GT,EQ,SO

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

Item Description

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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	SH
21 - 30	184
31	Rc

POWER® family

sliq *RA, RS, SH*

sliq. *RA, RS, SH*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
sliq	None	None	0	None
sliq.	None	None	1	LT,GT,EQ,SO

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

Item Description

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.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	SH
21 - 30	248
31	Rc

POWER® family

slliq *RA, RS, SH*

slliq. *RA, RS, SH*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
slli	None	None	0	None
slliq.	None	None	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>SH</i>	Specifies immediate value for shift amount.

Examples

- 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:


```
# Assume GPR 4 contains 0x9000 3000.
# Assume the MQ Register contains 0xFFFF FFFF.
slli 6,4,0x3
# GPR 6 now contains 0x8001 8007.
# The MQ Register now contains 0x8001 8004.
```
- 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 the MQ Register contains 0xFFFF FFFF.
slliq. 6,4,0x4
# GPR 6 now contains 0x0043 000F.
# The MQ Register contains 0x0043 000B.
# Condition Register Field 0 now contains 0x4.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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 **sllq** instruction is supported only in the POWER® family architecture.

Syntax

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	216
31	Rc

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
sllq	None	None	0	None
sllq.	None	None	1	LT,GT,EQ,SO

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

Item Description

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.
```

- 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 0xB004 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	152
31	Rc

POWER® family

slq *RA, RS, RB*
slq. *RA, RS, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
slq	None	None	0	None
slq.	None	None	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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 000B.
# Condition Register Field 0 now contains 0x4.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	24
31	Rc

PowerPC®

slw *RA, RS, RB*

slw. *RA, RS, RB*

POWER® family

sl *RA, RS, RB*

sl. *RA, RS, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
slw	None	None	0	None
slw.	None	None	1	LT,GT,EQ,SO
sl	None	None	0	None
sl.	None	None	1	LT,GT,EQ,SO

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

Item	Description
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:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 5 contains 0x0000 0005.
slw. 6,4,5
# GPR 6 now contains 0x0086 0000.
# Condition Register Field 0 now contains 0x4.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	S
11 - 15	A
16 - 20	B
21 - 30	794
31	Rc

POWER® family

srad *RA, RS, RB* (Rc=0)

srad. *RA, RS, RB* (Rc=1)

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 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. Shift amounts from 64 to 127 give a result of 64 sign bits in GPR *RA*, and cause XER[CA] to receive the sign bit of GPR *RS*.

Note that the **srad** instruction, followed by **addze**, can be 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

Item Description

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

Bits	Value
0 - 5	31
6 - 10	S
11 - 15	A
16 - 20	sh
21 - 29	413
30	sh
31	Rc

POWER® family

sradl *RA, RS, SH* (Rc=0)
sradl. *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 **sradl** instruction, followed by **addze**, can be used to divide quickly by 2^{**n} . The setting of the CA bit, by **sradl**, is independent of mode.

Other registers altered:

- Condition Register (CR0 field):
Affected: LT, GT, EQ, SO (if Rc = 1)
- XER:
Affected: CA

Parameters

Item	Description
<i>RA</i>	Specifies target general-purpose register for the result of the operation.
<i>RS</i>	Specifies source general-purpose register containing the operand for the shift operation.
<i>SH</i>	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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	SH
21 - 30	952
31	Rc

POWER® family

sraiq *RA, RS, SH*

sraiq. *RA, RS, SH*

Description

The **srai**q 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 fullword. 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 **srai**q instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
srai q	None	CA	0	None
srai q.	None	CA	1	LT,GT,EQ,SO

The two syntax forms of the **srai**q 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

Item Description

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.
```

- 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 concepts:

“addze or aze (Add to Zero Extended) instruction” on page 169

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	920
31	Rc

POWER® family

sraq *RA, RS, RB*

sraq. *RA, RS, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
sraq	None	CA	0	None
sraq.	None	CA	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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 0xB004 3000.
# Assume GPR 7 contains 0x0000 0004.
sraq. 6,4,7
# GPR 6 now contains 0xFB00 4300.
# The MQ Register now contains 0x0B00 4300.
# Condition Register Field 0 now contains 0x4.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	792
31	Rc

PowerPC®

sraw *RA, RS, RB*

sraw. *RA, RS, RB*

POWER® family

sra *RA, RS, RB*

sra. *RA, RS, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
sraw	None	CA	0	None
sraw.	None	CA	1	LT,GT,EQ,SO
sra	None	CA	0	None
sra.	None	CA	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 5 contains 0x0000 0004.
sraw. 6,4,5
# GPR 6 now contains 0xFB00 4300.
# Condition Register Field 0 now contains 0x8.
```

Related concepts:

“addze or aze (Add to Zero Extended) instruction” on page 169

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	SH
21 - 30	824
31	Rc

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
srawi	None	CA	0	None
srawi.	None	CA	1	LT,GT,EQ,SO
srai	None	CA	0	None
srai.	None	CA	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>SH</i>	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 concepts:

“addze or aze (Add to Zero Extended) instruction” on page 169

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	S
11 - 15	A
16 - 20	B
21 - 30	539
31	Rc

POWER® family

srd *RA, RS, RB* (Rc=0)

srd. *RA, RS, RB* (Rc=1)

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

Item	Description

<i>RA</i>	Specifies target general-purpose register for the result of the operation.

<i>RS</i>	Specifies source general-purpose register containing the operand for the shift operation.

<i>RB</i>	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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	665
31	Rc

POWER® family

sre *RA, RS, RB*
sre. *RA, RS, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
sre	None	None	0	None
sre.	None	None	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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 0x0009 0003.  
# The MQ Register now contains 0x0009 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 0xB004 3000.  
# Assume GPR 5 contains 0x0000 000F.  
sre. 6,4,5  
# GPR 6 now contains 0x0001 6008.  
# The MQ Register now contains 0x6001 6008.  
# 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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	921
31	Rc

POWER® family

srea *RA, RS, RB*

srea. *RA, RS, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
srea	None	CA	0	None
srea.	None	CA	1	LT,GT,EQ,SO

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

Item	Description
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 concepts:

“addze or aze (Add to Zero Extended) instruction” on page 169

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	729
31	Rc

POWER® family

sreq *RA, RS, RB*
sreq. *RA, RS, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
sreq	None	None	0	None
sreq.	None	None	1	LT,GT,EQ,SO

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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 300F.
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 0xB00 300F.
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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage

mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	SH
21 - 30	696
31	Rc

POWER® family

sriq *RA, RS, SH*

sriq. *RA, RS, SH*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
sriq	None	None	0	None
sriq.	None	None	1	LT,GT,EQ,SO

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

Item	Description
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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	SH
21 - 30	760
31	Rc

POWER® family

srliq *RA, RS, SH*
srliq. *RA, RS, SH*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
srliq	None	None	0	None
srliq.	None	None	1	LT,GT,EQ,SO

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

Item Description

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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26
The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	728
31	Rc

POWER® family

srlq *RA, RS, RB*
srlq. *RA, RS, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
srlq	None	None	0	None
srlq.	None	None	1	LT,GT,EQ,SO

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

Item	Description
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

- 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:


```
# Assume GPR 4 contains 0x9000 300F.
# Assume GPR 8 contains 0x0000 0024.
# Assume the MQ Register contains 0xFFFF FFFF.
srlq 6,4,8
# GPR 6 now contains 0x0FFF FFFF.
# The MQ Register remains unchanged.
```
- 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:


```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 8 contains 0x00000 0004.
# Assume the MQ Register contains 0xFFFF FFFF.
srlq. 6,4,8
# GPR 6 now holds 0xFB00 4300.
# The MQ Register remains unchanged.
# Condition Register Field 0 now contains 0x8.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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.

Note: The **srq** instruction is supported only in the POWER® family architecture.

Syntax

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	664
31	Rc

POWER® family

srq *RA, RS, RB*

srq. *RA, RS, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
srq	None	None	0	None
srq.	None	None	1	LT,GT,EQ,SO

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

Item Description

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 300F.
# 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 0xB000 300F.  
# Assume GPR 25 holds 0x0000 0004.  
srq. 6,4,8  
# GPR 6 now holds 0x0B00 0300.  
# The MQ Register now holds 0xFB00 0300.  
# Condition Register Field 0 now contains 0x4.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	536
31	Rc

PowerPC®

srw *RA, RS, RB*
srw. *RA, RS, RB*

POWER® family

sr *RA, RS, RB*
sr. *RA, RS, RB*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
srw	None	None	0	None
srw.	None	None	1	LT,GT,EQ,SO
sr	None	None	0	None
sr.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **sr** instruction, and the two syntax forms of the **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

Item Description

- 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

- 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:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 5 contains 0x0000 0024.
srw 6,4,5
# GPR 6 now contains 0x0000 0000.
```
- 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:

```
# 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.
```

Related concepts:

“addze or aze (Add to Zero Extended) instruction” on page 169

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point rotate and shift instructions” on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

stb (Store Byte) instruction

Purpose

Stores a byte of data from a general-purpose register into a specified location in memory.

Syntax

Bits	Value
0 - 5	38
6 - 10	RS
11 - 15	RA
16 - 31	D

Item	Description
stb	<i>RS, D(RA)</i>

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

Item	Description
<i>RS</i>	Specifies source general-purpose register of stored data.
<i>D</i>	Specifies a 16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

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

Bits	Value
0 - 5	39
6 - 10	RS
11 - 15	RA
16 - 31	D

Item	Description
stbu	<i>RS, D(RA)</i>

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

Item	Description
<i>RS</i>	Specifies source general-purpose register of stored data.
<i>D</i>	Specifies a 16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 6 contains 0x0000 0060.
# Assume GPR 16 contains the address of csect data[rw].
.csect text[pr]
stbu 6,buffer(16)
# GPR 16 now contains the address of buffer.
# 0x60 is stored at the address of buffer.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	247
31	/

Item	Description
stbux	<i>RS, RA, RB</i>

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

Item	Description
<i>RS</i>	Specifies source general-purpose register of stored data.
<i>RA</i>	Specifies source general-purpose register for EA calculation and possible address update.
<i>RB</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

stbx (Store Byte Indexed) instruction

Purpose

Stores a byte from a general-purpose register into a specified location in memory.

Syntax

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	215
31	/

Item	Description
stbx	<i>RS, RA, RB</i>

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

Item	Description
<i>RS</i>	Specifies source general-purpose register of stored data.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage

mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

std (Store Double Word) instruction

Purpose

Store a doubleword of data from a general purpose register into a specified memory location.

Syntax

Bits	Value
0 - 5	62
6 - 10	RS
11 - 15	RA
16 - 29	DS
30 - 31	0

PowerPC 64

std *RS, Disp(RA)*

Description

The **std** instruction stores a doubleword in storage from the source general-purpose register (GPR) *RS* into the specified location in memory referenced by the effective address (EA).

DS is a 14-bit, signed two's complement number, which is sign-extended to 64 bits, and then multiplied by 4 to provide a displacement *Disp*. If GPR *RA* is not 0, the EA is the sum of the contents of GPR *RA* and *Disp*. If GPR *RA* is 0, then the EA is *Disp*.

Parameters

Item	Description

<i>RS</i>	Specifies the source general-purpose register containing data.

<i>Disp</i>	Specifies a 16-bit signed number that is a multiple of 4. The assembler divides this number by 4 when generating the instruction.

<i>RA</i>	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

Bits	Value
0 - 5	31
6 - 10	S
11 -- 15	A
16 - 20	B
21 - 30	214
31	1

POWER® family
stdcx. *RS, RA, RB*

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 doubleword 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 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

Item	Description
<i>RS</i>	Specifies source general-purpose register of stored data.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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.

stdu (Store Double Word with Update) instruction

Purpose

Store a doubleword of data from a general purpose register into a specified memory location. Update the address base.

Note: This instruction should only be used on 64-bit PowerPC processors running a 64-bit application.

Syntax

Bits	Value
0 - 5	62
6 - 10	RS
11 - 15	RA
16 - 29	DS
30 - 31	0b01

PowerPC 64

stdu *RS, Disp(RA)*

Description

The **stdu** instruction stores a doubleword in storage from the source general-purpose register (GPR) *RS* into the specified location in memory referenced by the effective address (EA).

DS is a 14-bit, signed two's complement number, which is sign-extended to 64 bits, and then multiplied by 4 to provide a displacement *Disp*. If GPR *RA* is not 0, the EA is the sum of the contents of GPR *RA* and *Disp*. If GPR *RA* is 0, then the EA is *Disp*.

If GPR *RA* = 0, the instruction form is invalid.

Parameters

Item	Description
<i>RS</i>	Specifies the source general-purpose register containing data.
<i>Disp</i>	Specifies a 16-bit signed number that is a multiple of 4. The assembler divides this number by 4 when generating the instruction.
<i>RA</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

stdux (Store Double Word with Update Indexed) instruction

Purpose

Store a doubleword of data from a general purpose register into a specified memory location. Update the address base.

Syntax

Bits	Value
0 - 5	31
6 - 10	S
11 - 15	A
16 - 20	B
21 - 30	181
31	0

POWER® family

stdux *RS, RA, RB*

Description

The **stdux** instruction stores a doubleword in storage from the source general-purpose register (GPR) *RS* into the location in storage specified by the effective address (EA).

The EA is the sum of the contents of GPR *RA* and *RB*. GRP *RA* is updated with the EA.

If *rA* = 0, the instruction form is invalid.

Parameters

Item	Description
<i>RS</i>	Specifies the source general-purpose register containing data.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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.

stdx (Store Double Word Indexed) instruction

Purpose

Store a doubleword of data from a general purpose register into a specified memory location.

Syntax

Bits	Value
0 - 5	31
6 - 10	S
11 - 15	A
16 - 20	B
21 - 30	149
31	0

POWER® family

stdx *RS, RA, RB*

Description

The **stdx** instruction stores a doubleword 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

Item	Description
<i>RS</i>	Specifies the source general-purpose register containing data.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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

Bits	Value
0 - 5	54
6 - 10	FRS
11 - 15	RA
16 - 31	D

Item	Description
stfd	<i>FRS, D(RA)</i>

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

Item	Description
<i>FRS</i>	Specifies source floating-point register of stored data.
<i>D</i>	Specifies a16-bit signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	Specifies source general-purpose register for EA calculation.

Examples

The following code stores the contents of FPR 6 into a location in memory:

```
.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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

stfdu (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

Bits	Value
0 - 5	55
6 - 10	FRS
11 - 15	RA
16 - 31	D

Item	Description
stfdu	FRS, D(RA)

Description

The **stfdu** 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*.

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 **stfdu** instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

Item	Description
FRS	Specifies source 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 doubleword contents of FPR 6 into a location in memory and stores the address in GPR 4:

```
.csect data[rw]
buffer: .long 0,0
# Assume FPR 6 contains 0x4865 6C6C 6F20 776F.
# GPR 4 contains the address of csect data[rw].
.csect text[pr]
stfdu 6,buffer(4)
# buffer now contains 0x4865 6C6C 6F20 776F.
# GPR 4 now contains the address of buffer.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

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

Bits	Value
0 - 5	31
6 - 10	FRS
11 - 15	RA
16 - 20	RB
21 - 30	759
31	/

Item	Description
stfdux	<i>FRS, RA, RB</i>

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

Item	Description
<i>FRS</i>	Specifies source floating-point register of stored data.
<i>RA</i>	Specifies source general-purpose register for EA calculation and possible address update.
<i>RB</i>	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:

```
.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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

stfdx (Store Floating-Point Double Indexed) instruction

Purpose

Stores a doubleword of data in a specified location in memory.

Syntax

Bits	Value
0 - 5	31
6 - 10	FRS
11 - 15	RA
16 - 20	RB
21 - 30	727
31	/

Item	Description
stfdx	<i>FRS, RA, RB</i>

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

Item	Description
<i>FRS</i>	Specifies source floating-point register of stored data.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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]
stfdx 6,4,5
# 0x4865 6C6C 6F20 776F is now stored at the
# address buffer+8.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

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

Bits	Value
0 - 5	31
6 - 10	FRS
11 - 15	RA
16 - 20	RB
21 - 30	983
31	/

Item	Description
stfiwx	<i>FRS, RA, RB</i>

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

Item	Description
<i>FRS</i>	Specifies source floating-point register of stored data.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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]
stfiwx 6,4,5
# 6F20 776F is now stored at the
# address buffer+8.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

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

Bits	Value
0 - 5	60
6 - 10	FRS
11 - 15	RA
16 - 29	DS
30 - 31	00

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

Item Description

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 concepts:

“Ifqux (Load Floating-Point Quad with Update Indexed) instruction” on page 301

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

stfqu (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 **stfqu** instruction is supported only in the POWER2™ implementation of the POWER® family architecture.

Syntax

Bits	Value
0 - 5	61
6 - 10	FRS
11 - 15	RA
16 - 29	DS
30 - 31	01

POWER2™

stfqu *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* 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 **stfqu** instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

Item Description

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 concepts:

"lfqux (Load Floating-Point Quad with Update Indexed) instruction" on page 301

"Floating-point processor" on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

"Floating-point load and store instructions" on page 29

stfqux (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 **stfqux** instruction is supported only in the POWER2™ implementation of the POWER® family architecture.

Syntax

Bits	Value
0 - 5	31
6 - 10	FRS
11 - 15	RA
16 - 20	RB
21 - 30	951
31	Rc

POWER2™

stfqux *FRS, RA, RB*

Description

The **stfqux** 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 **stfqux** instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

Item Description

- 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 concepts:

“lfqux (Load Floating-Point Quad with Update Indexed) instruction” on page 301

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

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

Bits	Value
0 - 5	31
6 - 10	FRS
11 - 15	RA
16 - 20	RB
21 - 30	919
31	Rc

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

Item Description

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 concepts:

“lfqux (Load Floating-Point Quad with Update Indexed) instruction” on page 301

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

stfs (Store Floating-Point Single) instruction

Purpose

Stores a word of data from a floating-point register into a specified location in memory.

Syntax

Bits	Value
0 - 5	52
6 - 10	FRS
11 - 15	RA
16 - 31	D

Item	Description
stfs	<i>FRS</i> , <i>D</i> (<i>RA</i>)

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

Item	Description
<i>FRS</i>	Specifies floating-point register of stored data.
<i>D</i>	Specifies a 16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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 csect data[rw].
.csect text[pr]
stfs 6,buffer(4)
# buffer now contains 0x432B 6363.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

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

Bits	Value
0 - 5	53
6 - 10	FRS
11 - 15	RA
16 - 31	D

Item	Description
stfsu	<i>FRS</i> , <i>D</i> (<i>RA</i>)

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

Item	Description
<i>FRS</i>	Specifies floating-point register of stored data.
<i>D</i>	Specifies a 16-bit, signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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]
stfsu 6,buffer(4)
# GPR 4 now contains the address of buffer.
# buffer now contains 0x432B 6363.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

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

Bits	Value
0 - 5	31
6 - 10	FRS
11 - 15	RA
16 - 20	RB
21 - 30	695
31	/

Item	Description
stfsux	<i>FRS, RA, RB</i>

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*.

The **stfsux** instruction has one syntax form and does not affect the Floating-Point Status and Control Register or Condition Register Field 0.

Parameters

Item	Description
<i>FRS</i>	Specifies floating-point register of stored data.
<i>RA</i>	Specifies source general-purpose register for EA calculation and possible address update.
<i>RB</i>	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:

```
.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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

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

Bits	Value
0 - 5	31
6 - 10	FRS
11 - 15	RA
16 - 20	RB
21 - 30	663
31	/

Item	Description
stfsx	FRS, RA, RB

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

Item	Description
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 concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

sth (Store Half) instruction

Purpose

Stores a halfword of data from a general-purpose register into a specified location in memory.

Syntax

Bits	Value
0 - 5	44
6 - 10	RS
11 - 15	RA
16 - 31	D

Item	Description
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

Item	Description
RS	Specifies source general-purpose 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 bits 16-31 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]
sth 6,buffer(4)
# buffer now contains 0x3000.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 -20	RB
21 - 30	918
31	/

Item	Description
sthbrx	<i>RS, RA, RB</i>

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 **sthbrx** instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

Item	Description
<i>RS</i>	Specifies source general-purpose register of stored data.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 6 contains 0x9000 3456.
# Assume GPR 4 contains the address of buffer.
.csect text[pr]
sthbrx 6,0,4
# buffer now contains 0x5634.
```

Related concepts:

“Floating-point processor” on page 28

The floating-point processor provides instructions to perform arithmetic, comparison, and other operations.

“Floating-point load and store instructions” on page 29

sth (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

Bits	Value
0 - 5	45
6 - 10	RS
11 - 15	RA
16 - 31	D

Item	Description
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*.

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*.

The **sth** instruction has one syntax form and does not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

Item	Description
RS	Specifies source general-purpose 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 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:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 6 contains 0x0000 3456.
# Assume GPR 4 contains the address of csect data[rw].
.csect text[pr]
sth 6,buffer(4)
# buffer now contains 0x3456
# GPR 4 contains the address of buffer.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16	RB
21 - 30	439
31	/

Item	Description
sthux	<i>RS, RA, RB</i>

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*.

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

Item	Description
<i>RS</i>	Specifies source general-purpose register of stored data.
<i>RA</i>	Specifies source general-purpose register for EA calculation and possible address update.
<i>RB</i>	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:

```
.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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

sthx (Store Half Indexed) instruction

Purpose

Stores a halfword of data from a general-purpose register into a specified location in memory.

Syntax

Bits	Value
0 - 5	31
6 - 10	<i>RS</i>
11 - 15	<i>RA</i>
16 - 20	<i>RB</i>
21 - 30	407
31	/

Item	Description
sthx	<i>RS</i> , <i>RA</i> , <i>RB</i>

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

Item	Description

<i>RS</i>	Specifies source general-purpose register of stored data.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

stmw or stm (Store Multiple Word) instruction

Stores the contents of consecutive registers into a specified memory location.

Syntax

Bits	Value
0 - 5	47
6 - 10	RT
11 - 15	RA
16 - 31	D

PowerPC®

stmw *RS, D(RA)*

POWER® family
stm *RS, D(RA)*

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

Item	Description
<i>RS</i>	Specifies source general-purpose register of stored data.
<i>D</i>	Specifies a 16-bit signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

stq (Store Quad Word) instruction

Purpose

Store a quad-word of data from a general purpose register into a specified memory location.

Syntax

Bits	Value
0 - 5	62
6 - 10	RS
11 - 15	RA
16 - 29	DS
30 - 31	0b10

PowerPC 64

stq *RS, Disp(RA)*

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).

DS is a 14-bit, signed two's complement number, which is sign-extended to 64 bits, and then multiplied by 4 to provide a displacement *Disp*. If GPR *RA* is not 0, the EA is the sum of the contents of GPR *RA* and *Disp*. If GPR *RA* is 0, then the EA is *Disp*.

Parameters

Item Description

- RS* Specifies the source general-purpose register containing data. If *RS* is odd, the instruction form is invalid.
- Disp* Specifies a 16-bit signed number that is a multiple of 4. The assembler divides this number by 4 when generating the instruction.
- 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

stswi or stsi (Store String Word Immediate) instruction

Purpose

Stores consecutive bytes from consecutive registers into a specified location in memory.

Syntax

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	NB
21 - 30	725
31	/

PowerPC®

stswi *RS, RA, NB*

POWER® family

stsi *RS, RA, NB*

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 $\text{ceiling}(N/4)$, 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

Item	Description
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:

```
.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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point string instructions” on page 23

The Fixed-Point String instructions allow the movement of data from storage to registers or from registers to storage without concern for alignment.

stswx or stsx (Store String Word Indexed) instruction

Purpose

Stores consecutive bytes from consecutive registers into a specified location in memory.

Syntax

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	661
31	/

PowerPC®

stswx *RS, RA, RB*

POWER® family

stsx *RS, RA, RB*

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

Item	Description
<i>RS</i>	Specifies source general-purpose register of stored data.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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:

```
.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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point string instructions” on page 23

The Fixed-Point String instructions allow the movement of data from storage to registers or from registers to storage without concern for alignment.

stw or st (Store) instruction

Purpose

Stores a word of data from a general-purpose register into a specified location in memory.

Syntax

Bits	Value
0 - 5	36
6 - 10	RS
11 - 15	RA
16 - 31	D

PowerPC®

stw *RS, D(RA)*

POWER® family
st *RS, D(RA)*

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

Item	Description
<i>RS</i>	Specifies source general-purpose register of stored data.
<i>D</i>	Specifies a16-bit signed two's complement integer sign-extended to 32 bits for EA calculation.
<i>RA</i>	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[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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	662
31	/

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

Item	Description
<i>RS</i>	Specifies source general-purpose register of stored data.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 4 contains the address of buffer.
# Assume GPR 9 contains 0x0000 0000.
# Assume GPR 6 contains 0x1234 5678.
.csect text[pr]
stwbrx 6,4,9
# 0x7856 3412 is now stored at the address of buffer.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage

mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	150
31	1

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 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 undefined.

Parameters

Item	Description
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:  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 concepts:

"lwarx (Load Word and Reserve Indexed) instruction" on page 327

"Processing and storage" on page 9

The processor stores the data in the main memory and in the registers.

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

Bits	Value
0 - 5	37
6 - 10	RS
11 - 15	RA
16 - 31	D

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

Item Description

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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
212 - 30	183
31	/

PowerPC®

stwux *RS, RA, RB*

POWER® family

stux *RS, RA, RB*

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

Item	Description

<i>RS</i>	Specifies source general-purpose register of stored data.
<i>RA</i>	Specifies source general-purpose register for EA calculation and possible address update.
<i>RB</i>	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[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.
```

```
.csect text[pr]
stwx 6,4,23
# buffer+4 now contains 0x9000 3000.
# GPR 4 now contains the address of buffer+4.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store with update instructions” on page 23

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.

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

Bits	Value
0 - 5	31
6 - 10	RS
11 - 15	RA
16 - 20	RB
21 - 30	151
31	/

PowerPC®

stwx *RS, RA, RB*

POWER® family

stx *RS, RA, RB*

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

Item	Description
<i>RS</i>	Specifies source general-purpose register of stored data.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point load and store instructions” on page 22

The fixed point load instructions move information from a location addressed by the effective address (EA) into one of the GPRs.

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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21	OE
22 - 30	40
31	Rc

PowerPC®

subf	<i>RT, RA, RB</i>
subf.	<i>RT, RA, RB</i>
subfo	<i>RT, RA, RB</i>
subfo.	<i>RT, RA, RB</i>

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
subf	0	None	0	None
subf.	0	None	1	LT,GT,EQ,SO
subfo	1	SO,OV,CA	0	None
subfo.	1	SO,OV,CA	1	LT,GT,EQ,SO

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for EA calculation.
<i>RB</i>	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 2B00.
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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21	OE
22 - 30	8
31	Rc

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
subfc	0	CA	0	None
subfc.	0	CA	1	LT,GT,EQ,SO
subfco	1	SO,OV,CA	0	None
subfco.	1	SO,OV,CA	1	LT,GT,EQ,SO
sf	0	CA	0	None
sf.	0	CA	1	LT,GT,EQ,SO
sfo	1	SO,OV,CA	0	None
sfo.	1	SO,OV,CA	1	LT,GT,EQ,SO

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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.
subfc. 6,4,10
# GPR 6 now contains 0x8000 2B00.
```

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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	RB
21	OE
22 - 30	136
31	Rc

PowerPC®

```
subfe      RT, RA, RB
subfe.     RT, RA, RB
subfeo     RT, RA, RB
subfeo.    RT, RA, RB
```

POWER® family

sfe	<i>RT, RA, RB</i>
sfe.	<i>RT, RA, RB</i>
sfeo	<i>RT, RA, RB</i>
sfeo.	<i>RT, RA, RB</i>

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
subfe	0	CA	0	None
subfe.	0	CA	1	LT,GT,EQ,SO
subfeo	1	SO,OV,CA	0	None
subfeo.	1	SO,OV,CA	1	LT,GT,EQ,SO
sfe	0	CA	0	None
sfe.	0	CA	1	LT,GT,EQ,SO
sfeo	1	SO,OV,CA	0	None
sfeo.	1	SO,OV,CA	1	LT,GT,EQ,SO

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>RB</i>	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.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

subfic or sfi (Subtract from Immediate Carrying) instruction

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

Bits	Value
0 - 5	08
6 - 10	RT
11 - 15	RA
16 - 31	SI

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

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.
<i>SI</i>	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:

```
# Assume GPR 4 holds 0x9000 3000.  
subfic 6,4,0x00007000  
# GPR 6 now holds 0x7000 4000.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	///
21	OE
22 - 30	232
31	Rc

PowerPC®

subfme *RT, RA*
subfme. *RT, RA*
subfmeo *RT, RA*
subfmeo. *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'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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
subfme	0	CA	0	None
subfme.	0	CA	1	LT,GT,EQ,SO
subfmeo	1	SO,OV,CA	0	None
subfmeo.	1	SO,OV,CA	1	LT,GT,EQ,SO
sfme	0	CA	0	None
sfme.	0	CA	1	LT,GT,EQ,SO
sfmeo	1	SO,OV,CA	0	None
sfmeo.	1	SO,OV,CA	1	LT,GT,EQ,SO

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 effects 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

Item	Description
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'FFFFFFFF' and stores the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
```

```
# Assume the Carry bit is set to one.
```

```
subfme 6,4
```

```
# 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'FFFFFFFF', 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 the Carry bit is set to zero.
```

```
subfme. 6,4
```

```
# GPR 6 now contains 0x4FFB 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'FFFFFFFF', 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 one.
```

```
subfmeo 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 of the Fixed-Point Exception Register, and x'FFFFFFFF', 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 0xEFFF FFFF.
```

```
# Assume the Carry bit is set to zero.
```

```
subfmeo. 6,4
```

```
# GPR 6 now contains 0x0FFF FFFF.
```

Related concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

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

Bits	Value
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	///
21	OE
22 - 30	200
31	Rc

PowerPC®

subfze *RT, RA*
subfze. *RT, RA*
subfzeo *RT, RA*
subfzeo. *RT, RA*

POWER® family

sfze *RT, RA*
sfze. *RT, RA*
sfzeo *RT, RA*
sfzeo. *RT, RA*

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
subfze	0	CA	0	None
subfze.	0	CA	1	LT,GT,EQ,SO
subfzeo	1	SO,OV,CA	0	None
subfzeo.	1	SO,OV,CA	1	LT,GT,EQ,SO
sfze	0	CA	0	None
sfze.	0	CA	1	LT,GT,EQ,SO
sfzeo	1	SO,OV,CA	0	None
sfzeo.	1	SO,OV,CA	1	LT,GT,EQ,SO

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 effects 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

Item	Description

<i>RT</i>	Specifies target general-purpose register where result of operation is stored.

<i>RA</i>	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 0x6FFF D000.
```

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 0xB004 3000.  
# Assume the Carry bit is set to one.  
subfze. 6,4  
# GPR 6 now contains 0x4FFB D000.
```

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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

svc (Supervisor Call) instruction

Purpose

Generates a supervisor call interrupt.

Note: The **svc** instruction is supported only in the POWER® family architecture.

Syntax

Bits	Value
0 - 5	17
6 - 10	///
11 - 15	///
16 - 19	FLI
20 - 26	LEV
27 - 29	FL2
30	SA
31	LK

POWER® family

svc *LEV, FL1, FL2*

svcl *LEV, FL1, FL2*

Bits	Value
0 - 5	17
6 - 10	///
11 - 15	///
16 - 29	SV
30	SA
31	LK

Item	Description
svca	<i>SV</i>
svcla	<i>SV</i>

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' || LEV || b'00000'*, 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.

Item	Description		
Syntax Form	Link Bit (LK)	SVC Absolute Bit (SA)	Machine State Register Bits
svc	0	0	EE,PR,FE set to zero
svcl	1	0	EE,PR,FE set to zero
svca	0	1	EE,PR,FE set to zero
svcla	1	1	EE,PR,FE set to zero

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

Item Description

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 concepts:

“**cror** (Condition Register OR) instruction” on page 202

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“System call instruction” on page 20

The PowerPC® system call instructions generate an interrupt or the system to perform a service.

“Functional differences for POWER® family and PowerPC® instructions” on page 144

The POWER® family and PowerPC® instructions that share the same op code on POWER® family and PowerPC® platforms, but differ in their functional definition.

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

Bits	Value
0 - 5	31
6 - 9	///
10	L
11 - 15	///
16 - 20	///
21 - 30	598
31	/

PowerPC®
sync L

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 eieio (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

Item	Description
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:

```
# Assume that GPR 4 holds 0x0000 3000.  
dcbf 1,4  
sync  
# Wait for memory to be updated.
```

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

“.ei pseudo-op” on page 533

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

Bits	Value
0 - 5	31
6 - 10	TO
11 - 15	A
16 - 20	B
21 - 30	68
31	0

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:

TO bit	ANDed with Condition
0	Compares Less Than.
1	Compares Greater Than.
2	Compares Equal.
3	Compares Logically Less Than.
4	Compares Logically Greater Than.

Parameters

Item	Description
<i>TO</i>	Specifies <i>TO</i> bits that are ANDed with compare results.
<i>RA</i>	Specifies source general-purpose register for compare.
<i>RB</i>	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:

```
# 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 concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Fixed-point trap instructions” on page 25

Fixed-point trap instructions test for a specified set of conditions.

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

Bits	Value
0 - 5	02
6 - 10	TO
11 - 15	A
16 - 31	SIMM

PowerPC64

tdi *TO, RA, SIMM*

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:

TO bit	ANDed with Condition
0	Compares Less Than.
1	Compares Greater Than.
2	Compares Equal.
3	Compares Logically Less Than.
4	Compares Logically Greater Than.

Parameters

Item	Description
<i>TO</i>	Specifies <i>TO</i> bits that are ANDed with compare results.
<i>RA</i>	Specifies source general-purpose register for compare.
<i>SIMM</i>	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 concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Fixed-point trap instructions” on page 25

Fixed-point trap instructions test for a specified set of conditions.

tlbie or tlbi (Translation Look-Aside Buffer Invalidate Entry) instruction

Purpose

Makes a translation look-aside buffer entry invalid for subsequent address translations.

Note:

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

Bits	Value
0 - 5	31
6 - 9	///
10	L
11 - 15	///
16 - 20	RB
21 - 30	306
31	/

PowerPC®

tlbie *RB, L*

POWER® family

tlbi *RA, RB*

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:

Item	Description
<i>RB</i>	Specifies the source general-purpose register containing the EA for the search.
<i>L</i>	Specifies the page size.

The following parameters pertain to the POWER® family instruction, **tlbi**, only:

Item	Description
<i>RA</i>	Specifies the source general-purpose register for EA calculation and, if <i>RA</i> is not GPR 0, the target general-purpose register for operation.
<i>RB</i>	Specifies source general-purpose register for EA calculation.

Security

The **tlbie** and **tlbi** instructions are privileged.

Related concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

tlbld (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.

Note:

1. The **tlbld** 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

Bits	Value
0 - 5	31
6 - 10	///
11 - 15	///
16 - 20	RB
21 - 30	978
31	/

Item	Description
PowerPC 603 RISC Microprocessor	PowerPC 603 RISC Microprocessor
tlbld	<i>RB</i>

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

Item	Description
<i>RB</i>	Specifies the source general-purpose register for EA.

Security

The **tlbld** instruction is privileged.

Related concepts:

“tlbie or tlbi (Translation Look-Aside Buffer Invalidate Entry) instruction” on page 502

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

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.

Note:

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

Bits	Value
0 - 5	31
6 - 10	///
11 - 15	///
16 - 20	RB
21 - 30	1010
31	/

Item	Description
PowerPC 603 RISC Microprocessor	PowerPC 603 RISC Microprocessor
tlbli	<i>RB</i>

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.

Related concepts:

“tlbld (Load Data TLB Entry) instruction” on page 503

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

Item	Description

<i>RB</i>	Specifies the source general-purpose register for EA.

Security

The **tlbli** instruction is privileged.

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

Bits	Value
0 - 5	31
6 - 10	///
11 - 15	///
16 - 20	///
21 - 30	566
31	/

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 concepts:

“Processing and storage” on page 9

The processor stores the data in the main memory and in the registers.

tw or t (Trap Word) instruction

Purpose

Generates a program interrupt when a specified condition is true.

Syntax

Bits	Value
0-5	31
6-10	TO
11-15	RA
16-20	RB
21-30	4
31	/

PowerPC®

tw *TO, RA, RB*

POWER® family

t *TO, RA, RB*

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.

TO bit	ANDed with Condition
0	Compares Less Than.
1	Compares Greater Than.
2	Compares Equal.
3	Compares Logically Less Than.
4	Compares Logically Greater Than.

The **tw** and **t** instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

Item	Description
<i>TO</i>	Specifies <i>TO</i> bits that are ANDed with compare results.
<i>RA</i>	Specifies source general-purpose register for compare.
<i>RB</i>	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 concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Fixed-point trap instructions” on page 25

Fixed-point trap instructions test for a specified set of conditions.

twi or ti (Trap Word Immediate) instruction

Purpose

Generates a program interrupt when a specified condition is true.

Syntax

Bits	Value
0-5	03
6-10	<i>TO</i>
11-15	<i>RA</i>
16-31	<i>SI</i>

PowerPC®

twi *TO, RA, SI*

POWER® family

ti *TO, RA, SI*

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.

TO bit	ANDED with Condition
0	Compares Less Than.
1	Compares Greater Than.
2	Compares Equal.
3	Compares Logically Less Than.
4	Compares Logically Greater Than.

The **twi** and **ti** instructions have one syntax form and do not affect the Fixed-Point Exception Register or Condition Register Field 0.

Parameters

Item	Description
<i>TO</i>	Specifies <i>TO</i> bits that are ANDed with compare results.
<i>RA</i>	Specifies source general-purpose register for compare.
<i>SI</i>	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 concepts:

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

“Fixed-point trap instructions” on page 25

Fixed-point trap instructions test for a specified set of conditions.

xor (XOR) instruction

Purpose

XORs the contents of two general-purpose registers and places the result in another general-purpose register.

Syntax

Bits	Value
0-5	31
6-10	RS
11-15	RA
16-20	RB
21-30	316
31	Rc

Item	Description
xor	<i>RA, RS, RB</i>
xor.	<i>RA, RS, RB</i>

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.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
xor	None	None	0	None
xor.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>RB</i>	Specifies source general-purpose register for operation.

Examples

- 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.
```
- 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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

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

Bits	Value
0-5	26
6-10	RS
11-15	RA
16-31	UI

PowerPC®

xori *RA, RS, UI*

POWER® family

xoril *RA, RS, UI*

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>UI</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

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

Bits	Value
0-5	27
6-10	RS
11-15	RA
16-31	UI

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

Item	Description
<i>RA</i>	Specifies target general-purpose register where result of operation is stored.
<i>RS</i>	Specifies source general-purpose register for operation.
<i>UI</i>	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 concepts:

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Fixed-point logical instructions” on page 26

Fixed-Point logical instructions perform logical operations in a bit-wise fashion.

Pseudo-ops

The pseudo-ops reference information includes an overview of the assembler pseudo-ops.

This topic provides an overview of assembler pseudo-ops and reference information for all pseudo-ops.

Pseudo-ops overview

A pseudo-op is an instruction to the assembler.

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

The pseudo-ops are grouped by functionality.

Pseudo-ops can be related according to functionality into the following groups:

Data alignment:

A pseudo-op is used for data alignment.

The following pseudo-op is used in the data or text section of a program:

- **.align**

Data definition:

There are different pseudo-ops defined for data definition.

The following pseudo-ops are used to create data areas to be used by a program:

- **.byte**
- **.double**
- **.float**
- **.leb128**
- **.long**
- **.ptr**
- **.quad**

- **.short**
- **.space**
- **.string**
- **.uleb128**
- **.vbyte**

Storage definition:

The pseudo-op defined for layout of the data area.

The following pseudo-op defines the layout of data areas:

- **.dsect**

Addressing:

The pseudo-ops defined to register a base register.

The following pseudo-ops assign or dismiss a register as a base register:

- **.drop**
- **.using**

Related concepts:

“.drop pseudo-op” on page 528

“.using pseudo-op” on page 561

Assembler section definition:

The pseudo-ops used to define the sections of an assembly language program.

The following pseudo-ops define the sections of an assembly language program:

- **.comm**
- **.csect**
- **.lcomm**
- **.tc**
- **.toc**

Related concepts:

“.comm pseudo-op” on page 522

“.csect pseudo-op” on page 525

“.lcomm pseudo-op” on page 541

“.tc pseudo-op” on page 558

“.toc pseudo-op” on page 559

External symbol definition:

The pseudo-ops used to define an global variable.

The following pseudo-ops define a variable as a global variable or an external variable (variables defined in external modules):

- **.extern**
- **.globl**
- **.weak**

Related concepts:

“.extern pseudo-op” on page 535

“.globl pseudo-op” on page 538

“.xline pseudo-op” on page 567

“Fixed-point processor” on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

“Branch processor” on page 19

The branch processor instructions include the branch instructions, Condition Register field and logical instructions.

Static symbol definition:

The pseudo-ops used to define a static symbol.

The following pseudo-op defines a static symbol:

- **.lglobl**

Support for calling conventions:

The pseudo-op used to define a debug traceback.

The following pseudo-op defines a debug traceback tag for performing tracebacks when debugging programs:

- **.tbttag**

Miscellaneous:

The pseudo-ops used to perform miscellaneous functions.

The following pseudo-ops perform miscellaneous functions:

Item	Description
.comment	Adds arbitrary information to the .comment section.
.except	Adds entries to the .except section.
.hash	Provides type-checking information.
.info	Adds arbitrary information to the .info section and generates a C_INFO symbol.
.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 pseudo-ops used to provide additional information to the symbolic debugger.

The following pseudo-ops provide additional information which is required by the symbolic debugger (dbx):

- **.bb**
- **.bc**
- **.bf**

- **.bi**
- **.bs**
- **.eb**
- **.ec**
- **.ef**
- **.ei**
- “.ei pseudo-op” on page 533
- **.file**
- **.function**
- **.line**
- **.stabx**
- **.xline**

Related concepts:

“.bb pseudo-op” on page 518

“.bi pseudo-op” on page 520

“.bs pseudo-op” on page 520

“.eb pseudo-op” on page 531

“.ef pseudo-op” on page 532

“.file pseudo-op” on page 536

“.function pseudo-op” on page 537

“.line pseudo-op” on page 543

“.stabx pseudo-op” on page 555

“.xline pseudo-op” on page 567

Target environment indication:

The pseudo-op used to define the target environment.

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:

Item	Description
<i>Name</i>	Any valid label.
<i>QualName</i>	<i>Name</i> {StorageMappingClass}
	An optional <i>Name</i> followed by an optional storage-mapping class enclosed in braces or brackets.
<i>Register</i>	A general-purpose register. <i>Register</i> is an expression that evaluates to an integer between 0 and 31, inclusive.
<i>Number</i>	An expression that evaluates to an integer.
<i>Expression</i>	Unless otherwise noted, the <i>Expression</i> variable signifies a relocatable constant or absolute expression.

Item	Description
<i>FloatingConstant</i>	A floating-point constant.
<i>StringConstant</i>	A string constant.
[]	Brackets enclose optional operands except in the “.csect pseudo-op” on page 525 and “.tc pseudo-op” on page 558, which require brackets in syntax.

.align pseudo-op

Purpose

Advances the current location counter until a boundary specified by the *Number* parameter is reached.

Syntax

Item	Description
.align	<i>Number</i>

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

Item	Description
<i>Number</i>	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.

Examples

The following example demonstrates the use of the **.align** pseudo-op:

```
.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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.comm pseudo-op” on page 522

“.csect pseudo-op” on page 525

“.double pseudo-op” on page 527

“.float pseudo-op” on page 537

“.long pseudo-op” on page 544

.bb pseudo-op

Purpose

Identifies the beginning of an inner block and provides information specific to the beginning of an inner block.

Syntax

Item	Description
.bb	<i>Number</i>

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

Item	Description
<i>Number</i>	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:

```
.bb 5
```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.eb pseudo-op” on page 531

.bc pseudo-op

Purpose

Identifies the beginning of a common block and provides information specific to the beginning of a common block.

Syntax

Item	Description
.bc	<i>StringConstant</i>

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

Item	Description
<i>StringConstant</i>	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 concepts:

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

.bf pseudo-op

Purpose

Identifies the beginning of a function and provides information specific to the beginning of a function.

Syntax

Item	Description
.bf	<i>Number</i>

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

Item	Description
<i>Number</i>	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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.ef pseudo-op” on page 532

“.function pseudo-op” on page 537

.bi pseudo-op

Purpose

Identifies the beginning of an included file and provides information specific to the beginning of an included file.

Syntax

Item	Description
.bi	<i>StringConstant</i>

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

Item	Description
<i>StringConstant</i>	Represents the name of the original source file.

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.line pseudo-op” on page 543

“.ei pseudo-op” on page 533

.bs pseudo-op

Purpose

Identifies the beginning of a static block and provides information specific to the beginning of a static block.

Syntax

Item	Description
.bs	<i>Name</i>

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

Item	Description
<i>Name</i>	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=G5TW01:3=Cc5,0,5;;,0,40;;",0x258,133,8
.es
```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.comm pseudo-op” on page 522

“.lcomm pseudo-op” on page 541

.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

Item	Description
<i>Expression</i>	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]
l 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 concepts:

“.align pseudo-op” on page 517

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.string pseudo-op” on page 556

“.vbyte pseudo-op” on page 565

“.llong pseudo-op” on page 544

.comm pseudo-op

Purpose

Defines an uninitialized block of storage called a common block.

Syntax

Item	Description
.comm	<i>Qualname</i> , <i>Expression</i> , [<i>Number</i> [<i>Visibility</i>]]

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. If *Visibility* is specified, *Number* can be omitted.

Description

The **.comm** pseudo-op defines a common block, which is an uninitialized block of storage. The *QualName* parameter specifies the name of the common block. *QualName* is defined as a global symbol.

The *Expression* parameter specifies the size of the common block in bytes.

Note: Symbols with the TD *StorageMappingClass* are conventionally no longer than a pointer. (A pointer is 4 bytes long in 32-bit mode and 8 bytes long in 64-bit mode.)

The valid values for the *StorageMappingClass* parameter are BS, RW, TD, UC, and UL. These values are explained in the description of the **.csect** pseudo-op. If any other value is used for the *StorageMappingClass* parameter, the default value RW is used, and if the **-w** flag was used, a warning message is reported.

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 link 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.

The visibility of the common block can be specified by using the *Visibility* parameter.

Parameters

Item	Description
<i>Qualname</i>	Specifies the name and storage mapping class of the common block. If the <i>StorageMappingClass</i> part of the parameter is omitted, the default value RW is used. Valid <i>StorageMappingClass</i> values for a common block are RW, TD, UC, BS, and UL.
<i>Expression</i>	Specifies the absolute expression that gives the length of the specified common block in bytes.
<i>Number</i>	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.
<i>Visibility</i>	Specifies the visibility of the symbol. Valid values are <i>exported</i> , <i>protected</i> , <i>hidden</i> , and <i>internal</i> . Symbol visibilities are used by linker.

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
# global

# 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)
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
cc -o td2 td2.c mod2.o

```

d. Running td2 will cause the following to be printed:

```

t_data is 1248

```

3. The following example shows how to specify visibility for a symbol:

```

.comm block, 1024, , exported
# block is an uninitialized block of storage 1024 bytes
# long. At link time, block is exported.

```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.align pseudo-op” on page 517

“.csect pseudo-op” on page 525

“.globl pseudo-op” on page 538

“.lcomm pseudo-op” on page 541

“.long pseudo-op” on page 544

“Visibility of symbols” on page 46

| .comment pseudo-op

| Purpose

| Adds arbitrary information to the comment section of the output file.

| Syntax

| `.comment StringConstant | Number`

| Description

| The `.comment` pseudo-op allows arbitrary information to the output object file. This information is added to the `.comment` section.

| The `.comment` pseudo-op is generated by a cascade compiler. The *StringConstant* or *Number* parameter is not interpreted by the assembler and it is not preserved by the `ld` command.

| Parameters

| **StringConstant**

| Specifies an arbitrary string to be added to the `.comment` section of the output file.

| **Number**

| Specifies an arbitrary byte value to be added to the `.comment` section of the output file.

.csect pseudo-op

Purpose

Groups code or data into a csect (control section) and gives that csect a name, a storage-mapping class, and an alignment.

Syntax

Item	Description
<code>.csect</code>	<i>QualName</i> [, <i>Number</i>]

where *QualName* = [*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 csect pseudo-op has an associated storage-mapping class. The storage-mapping class determines the object section in which the csect pseudo-op is grouped. The `.text` section usually contains read-only data, such as instructions or constants. The `.data`, `.bss`, `.tdata`, and `.tbss` sections contain read/write data. The storage-mapping classes for `.bss` and `.tbss` must be used with the `.comm` and `.lcomm` pseudo-ops, not the `.csect` pseudo-op.

The storage-mapping class also indicates what kind of data should be contained within the csect. 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 pseudo-op is associated with one of the following storage-mapping classes. Storage-mapping class identifiers are not case-sensitive. The storage-mapping class identifiers are listed in groups for the `.data`, `.tbss`, `.tdata`, `.text`, and `.tss` object file 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)

.tdata Section Storage-Mapping Classes

TL	Initialized thread-local storage. Identifies a csect that is instantiated at run time for each thread in the program.

.tbss Section Storage-Mapping Classes

UL	Uninitialized thread-local storage. Identifies a csect that is instantiated at run time for each thread in the program.

- All of the csects 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 csect is relocated as a body.
- csects with no specified name (*Name*) are identified with their storage-mapping class, and there can be an unnamed csect 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]) csect 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 csects defined with the **.csect** pseudo-op are of type SD (Section Definition). The **.comm** or **.lcomm** pseudo-ops can also be used to define csects of type CM. No external label can be defined in a csect of type CM.
- Do not label **.csect** statements. The **.csect** may be referred to by its *QualName*, and labels may be placed on individual elements of the **.csect**.

Parameters

Item	Description
<i>Number</i>	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 <i>Number</i> parameter for the .align pseudo-op. Alignment occurs at the beginning of the csect. Elements of the csect are not individually aligned. The <i>Number</i> parameter is optional. If it is not specified, the default value is 2.
<i>QualName</i>	Specifies a <i>Name</i> and <i>StorageMappingClass</i> for the csect. If <i>Name</i> is not given, the csect is identified with its <i>StorageMappingClass</i> . If neither the <i>Name</i> nor the <i>StorageMappingClass</i> are given, the csect is unnamed and has a storage-mapping class of [PR]. If the <i>Name</i> is specified, the <i>StorageMappingClass</i> must also be specified.

Examples

The following example defines three csects:

```
# A csect of name proga with Program Code Storage-Mapping Class.
.csect proga[PR]
lh      30,0x64(5)
# A csect of name pdata_ with Read-Only Storage-Mapping Class.
.csect pdata_[R0]
l1:     .long 0x7782
l2:     .byte 'a','b','c','d','e'
.csect [RW],3      # An unnamed csect with Read/Write
                  # Storage-Mapping Class and doubleword
                  # alignment.

.float -5
```

Related concepts:

- “Pseudo-ops overview” on page 512
- A pseudo-op is an instruction to the assembler.
- “.comm pseudo-op” on page 522
- “.globl pseudo-op” on page 538
- “.long pseudo-op” on page 544
- “.align pseudo-op” on page 517

.double pseudo-op

Purpose

Stores a double floating-point constant at the next fullword location.

Syntax

Item	Description
.double	<i>FloatingConstant</i> [, <i>FloatingConstant</i> ...]

Description

The **.double** pseudo-op assembles double floating-point constants into consecutive fullwords.

Parameters

Item	Description
<i>FloatingConstant</i>	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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.float pseudo-op” on page 537

.drop pseudo-op

Purpose

Stops using a specified register as a base register.

Syntax

Item	Description
.drop	<i>Number</i>

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

Item	Description
<i>Number</i>	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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.using pseudo-op” on page 561

.dsect pseudo-op

Purpose

Identifies the beginning or the continuation of a dummy control section.

Syntax

Item	Description
.dsect	<i>Name</i>

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

Item	Description
<i>Name</i>	Specifies a dummy control section.

Examples

1. The following example demonstrates the use of the **.dsect** pseudo-op:

```
.dsect data1
d1:    .long 0

                # 1 Fullwordd2:    .short 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    # 15 bytes

        .align 3                #Align to a double word.
d4:    .space 64                #Space 64 bytes

.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:

```
.csect    foo_data[RW]
.long    0xaa
.short    10
.short    20
.globl    .foo_pm[PR]
.csect    .foo_pm[PR]
.extern   l1
.using    TOC[TC0], 2
l 7, T.foo_data
b 11
br
.toc
T.foo_data: .tc foo_data[TC], foo_data[RW]
```

- 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 11
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
```

```

11:    .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:

```

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 cl.o cl.s
cc -o mm mm.c foo_pm.o bar_pm.o cl.o

```

f. The following is printed if mm is executed:

```

when sw is 2, t_data is 0xaa
when sw is 1, t_data is 0xbb

```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.csect pseudo-op” on page 525

“.using pseudo-op” on page 561

.eb pseudo-op

Purpose

Identifies the end of an inner block and provides additional information specific to the end of an inner block.

Syntax

Item	Description
.eb	<i>Number</i>

Description

The **.eb** pseudo-op identifies the end of an inner block and provides symbol table information necessary when using the symbolic debugger.

The **.eb** pseudo-op has no other effect on assembly and is customarily inserted by a compiler.

Parameters

Item	Description
<i>Number</i>	Specifies a line number in the original source file on which the inner block ends.

Examples

The following example demonstrates the use of the **.eb** pseudo-op:

```
.eb 10
```

Related concepts:

“Fixed-point arithmetic instructions” on page 24

The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

“.bb pseudo-op” on page 518

.ec pseudo-op

Purpose

Identifies the end of a common block and provides additional information specific to the end of a common block.

Syntax

```
.ec
```

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 concepts:

“.bc pseudo-op” on page 518

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

.ef pseudo-op

Purpose

Identifies the end of a function and provides additional information specific to the end of a function.

Syntax

Item	Description
<code>.ef</code>	<i>Number</i>

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

Item	Description
<i>Number</i>	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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“`.bf` pseudo-op” on page 519

`.ei` pseudo-op

Purpose

Identifies the end of an included file and provides additional information specific to the end of an included file.

Syntax

```
.ei
```

Description

The `.ei` pseudo-op identifies the end of an included file and provides symbol table information necessary when using the symbolic debugger.

The `.ei` pseudo-op has no other effect on assembly and is customarily inserted by a compiler.

Examples

The following example demonstrates the use of the `.ei` pseudo-op:

```
.ei "file.s"
```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“`.bi` pseudo-op” on page 520

.es pseudo-op

Purpose

Identifies the end of a static block and provides additional information specific to the end of a static block.

Syntax

.es

Description

The **.es** pseudo-op identifies the end of a static block and provides symbol table information necessary when using the symbolic debugger.

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:

```
.lcomm cgdat, 0x2b4
.csect .text[PR]
.bs cgdat
.stabx "ONE:1=Ci2,0,4;",0x254,133,0
.stabx "TWO:S2=G5TW01:3=Cc5,0,5;,0,40;;",0x258,133,8
.es
```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.bs pseudo-op” on page 520

| .except pseudo-op

| Purpose

| Adds entries to the **.except** section of the output file, and adds a reference to the information from the specified symbol.

| Syntax

| **.except** *Name*, *StringConstant* | *Number1*, *Number2*

| Description

| The **.except** section contains information about the reason for the trap instruction creation. The **.except** pseudo-op must be used before the corresponding trap instruction in the source program. A symbol can have multiple **.except** pseudo-ops and these pseudo-ops are combined into a group of exceptions that are associated with the symbol.

| Parameters

| *Name*

| The name of the symbol to be associated with the **.except** entry. This name must appear in an **.globl** or **.lglobl** statement.

| ***StringConstant***
 | Name of a language. Valid language names are listed in the documentation of the `.source` pseudo-op
 | statement.

| ***Number1***
 | Expression denoting a language. For valid values for this parameter, see the `/usr/include/
 | storclass.h` header file.

| ***Number2***
 | The reason for the `.except` entry. The value cannot be 0.

.extern pseudo-op

Purpose

Declares a symbol as an external symbol that is defined in another file.

Syntax

Item	Description
<code>.extern</code>	<i>Name</i> [, <i>Visibility</i>]

Description

The `.extern` pseudo-op identifies the *Name* value as a symbol that is defined in another source file, and the *Name* parameter becomes an external symbol. Any external symbols used but 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

Item	Description
<i>Name</i>	Specifies the name of the symbol to be declared as an external symbol. <i>Name</i> can be a <i>Qualname</i> . A <i>Qualname</i> parameter specifies the <i>Name</i> and <i>StorageMappingClass</i> values for the control section.
<i>Visibility</i>	Specifies the visibility of symbol. Valid visibility values are <i>exported</i> , <i>protected</i> , <i>hidden</i> , and <i>internal</i> . Symbol visibilities are used by the linker.

Examples

The following example demonstrates the use of the `.extern` pseudo-op:

```
.extern proga[PR]
.toc
T.proga: .tc proga[TC],proga[PR]
```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“`.csect` pseudo-op” on page 525

“`.globl` pseudo-op” on page 538

“Visibility of symbols” on page 46

.file pseudo-op

Purpose

- | Identifies the file name of the source file and compiler-related information.

Syntax

```
| .file StringConstant  
| .file StringConstant, StringConstant1  
| .file StringConstant, [StringConstant1], StringConstant2  
| .file StringConstant, [StringConstant1], [StringConstant2], StringConstant3
```

Description

- | The .file pseudo-op provides the symbol table information to the symbolic debugger and the **ld** command. *StringConstant* is the file name, and it is used as the name of an auxiliary symbol `x_ftype == XTY_FN`. If *StringConstant1*, *StringConstant2*, and *StringConstant3* are specified, these values are added to the symbol table as the compiler time stamp with the `x_ftype` symbol set to `XTY_CT`; compiler version with the `x_ftype` symbol set to `XTY_CV`; and compiler-provided information with the `x_ftype` symbol set to `XTY_CD`. The .file pseudo-op does not make any other changes to the assembler operation and the pseudo op is added to the object code by a cascade compiler.
- | If the .file pseudo-op is not specified in the source code, the assembler processes the program considering the .file pseudo-op as the first statement. The assembler creates the .file pseudo op as the first statement by adding an entry in the symbol table with the source program name as the file name. If the source program is the standard input, the file name is `noname`. The assembler listing does not have this inserted entry.

Parameters

- | **StringConstant**
| A string that specifies the file name.
- | **StringConstant1**
| A string that specifies the compiler version.
- | **StringConstant2**
| A string that specifies the compiler time stamp.
- | **StringConstant3**
| A string that specifies arbitrary compiler information.

Examples

- | 1. Specify a file name.
| .file "myfile.c"
- | 2. Specify name, version, time stamp, and compiler information.
| .file "myfile.c", "Version 99", "01 Jan 2099", "no options"
- | 3. Specify file name and time stamp, but do not specify the version and compiler information.
| .file "myfile.c", , "Jan 1, 2099"

Related concepts:

"Pseudo-ops overview" on page 512
A pseudo-op is an instruction to the assembler.

.float pseudo-op

Purpose

Assembles floating-point constants.

Syntax

Item	Description
.float	<i>FloatingConstant</i> [, <i>FloatingConstant</i>]

Description

The **.float** assembles floating-point constants into consecutive fullwords.

Parameters

Item	Description
<i>FloatingConstant</i>	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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

.function pseudo-op

Purpose

Identifies a function and provides additional information specific to the function.

Syntax

Item	Description
.function	<i>Name</i> , <i>Expression1</i> , <i>Expression2</i> , <i>Expression3</i> [, <i>Expression4</i>]

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

Item	Description
<i>Name</i>	Represents the function <i>Name</i> and should be defined as a symbol or control section (csect) <i>Qualname</i> in the current assembly. (A <i>Qualname</i> specifies a <i>Name</i> and <i>StorageMappingClass</i> for the control section.)
<i>Expression1</i>	Represents the top of the function.
<i>Expression2</i>	Represents the storage mapping class of the function.
<i>Expression3</i>	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).

Item	Description
<i>Expression4</i>	Represents the size of the function (in bytes). This parameter must be an absolute expression. This parameter is optional. Note: If the <i>Expression4</i> 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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.bf pseudo-op” on page 519

“.ef pseudo-op” on page 532

“.file pseudo-op” on page 536

.globl pseudo-op

Purpose

Declares a symbol to be a global symbol.

Syntax

Item	Description
.globl	<i>Name</i> [, <i>Visibility</i>]

Description

The **.globl** pseudo-op indicates that the symbol name is a global symbol and can be referenced by other files during linking. The **.extern**, **.weak**, or **.comm** pseudo-op can also be used to make a global symbol.

The visibility of the global symbol can be specified with the *Visibility* parameter.

Parameters

Item	Description
<i>Name</i>	Specifies the name of the label or symbol to be declared global. <i>Name</i> can be a <i>Qualname</i> . (A <i>Qualname</i> specifies a <i>Name</i> and <i>StorageMappingClass</i> for the control section.)
<i>Visibility</i>	Specifies the visibility of the symbol. Valid visibility values are <i>exported</i> , <i>hidden</i> , <i>internal</i> , and <i>protected</i> . Symbol visibilities are used by the linker.

Examples

The following example illustrates the use of the **.globl** pseudo-op:

```
.globl main
main:
    .csect data[rw]
    .globl data[rw], protected

# data[RW] is a global symbol and is to be exported with
# protected visibility at link time.
```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.comm pseudo-op” on page 522

“.extern pseudo-op” on page 535

“.weak pseudo-op” on page 566

“Visibility of symbols” on page 46

.hash pseudo-op

Purpose

Associates a hash value with an external symbol.

Syntax

Item	Description
.hash	<i>Name</i> , <i>StringConstant</i>

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

Item	Description
<i>Name</i>	Represents a symbol. Because this should be an external symbol, <i>Name</i> should appear in an .extern or .globl statement.
<i>StringConstant</i>	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], "0000A9375C1F51C2DCF0"
.hash a[pr], "ff0a2cc12365de30" # warning may report
.hash e[pr], "00002020202051C2DCF0"
```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.extern pseudo-op” on page 535

“.globl pseudo-op” on page 538

Related information:

XCOFF

| .info pseudo-op

| Purpose

| Adds arbitrary information to the .info section of the output file, and generates a C_INFO symbol with a specified name.

| Syntax

```
| .info Name, Number0 [, Number] ...
| .info , Number [, Number]...
```

| Description

| The .info pseudo-op adds information to the .info section of the output object file. The **Number0** value and other **Number** values are added to the .info section as word values. If the **Name** parameter is specified, a C_INFO symbol with the specified name is generated, whose value is the offset of the word after Number0 that is in the .info section. The Number0 value is used by the **ld** command to determine the length of the data that is associated with the C_INFO symbol.

| The `.info` pseudo-op is used by a cascade compiler. The assembler and the `ld` command do not interpret
| the constants.

| Parameters

| *Name*

| Specifies an arbitrary string to be added to the `.comment` section of the output file.

| *Number0*

| Specifies the aggregate length of the information to be added in bytes to the `.info` section of the
| output file.

| *Number*

| Specifies arbitrary words to be added to the `.info` section of the output file.

.lcomm pseudo-op

Purpose

Defines a local uninitialized block of storage.

Syntax

Item	Description
<code>.lcomm</code>	<i>Name1</i> , <i>Expression1</i> [, <i>Sectname</i> [, <i>Expression2</i>]]

Description

The `.lcomm` psuedo-op defines a local, uninitialized block of storage. If the *Sectname* parameter is a QualName operand with a **StorageMappingClass** class of UL, the storage is for thread-local variables.

The `.lcomm` pseudo-op is used for data that will probably not be accessed in other source files.

The *Name1* parameter is a label at the beginning of the block of storage. The location counter for this block of storage is incremented by the value of the *Expression1* parameter. A specific storage block can be specified with the *Sectname* parameter. If the *Sectname* parameter is a QualName operand with a **StorageMappingClass** class of UL, the storage block is assigned to the `.tbss` section. Otherwise, the storage block is assigned to the `.bss` section. If *Sectname* is not specified, an unnamed storage block is used.

The alignment of the storage block can be specified with the *Expression2* parameter. If *Expression2* is omitted, the block of storage is aligned on a half-word boundary.

Parameters

Item	Description
<i>Name1</i>	Label on the block of storage. <i>Name1</i> does not appear in the symbol table unless it is the operand of a <code>.globl</code> statement.
<i>Expression1</i>	An absolute expression specifying the length of the block of storage.
<i>Sectname</i>	Optional csect name. If <i>Sectname</i> is omitted, an unnamed block of storage with a storage-mapping class of BS is used. If <i>Sectname</i> is a QualName, StorageMappingClass can be BS or UL. If <i>Sectname</i> is a symbol name, the default storage-mapping class BS is used. The same <i>Sectname</i> can be used with multiple <code>.lcomm</code> statements. The blocks of storage specified by the <code>.lcomm</code> statements are combined into a single csect .
<i>Expression2</i>	An absolute expression specifying the log base 2 of the desired alignment. If <i>Expression2</i> is omitted, a value of 2 is used, resulting in a half-word alignment.

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".
```

3. To define a local block of thread-local storage:

```
.lcomm tls1,32,tls_static[UL],3
    # tls1 is a label on a block of thread-local storage 32 bytes
    # long aligned on a doubleword boundary. The name of the block of
    # storage is tls_static[UL].
```

Related concepts:

"Pseudo-ops overview" on page 512

A pseudo-op is an instruction to the assembler.

".comm pseudo-op" on page 522

.lglobl pseudo-op

Purpose

Provides a means to keep the information of a static name in the symbol table.

Syntax

Item	Description
.lglobl	<i>Name</i>

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

Item	Description
<i>Name</i>	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 **.lglobl** 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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.function pseudo-op” on page 537

“.globl pseudo-op” on page 538

.line pseudo-op

Purpose

Identifies a line number and provides additional information specific to the line number.

Syntax

Item	Description
.line	<i>Number</i>

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

Item	Description
<i>Number</i>	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
.stabx "hello:f-1",0,142,0
.bf 2
.line 1
.line 2

```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.bi pseudo-op” on page 520

“.bf pseudo-op” on page 519

“.function pseudo-op” on page 537

.long pseudo-op

Purpose

Assembles expressions into consecutive fullwords.

Syntax

Item	Description
.long	<i>Expression[,Expression,...]</i>

Description

The **.long** pseudo-op assembles expressions into consecutive fullwords. Fullword alignment occurs as necessary.

Parameters

Item	Description
<i>Expression</i>	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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.byte pseudo-op” on page 521

“.short pseudo-op” on page 552

“.vbyte pseudo-op” on page 565

.llong pseudo-op

Purpose

Assembles expressions into consecutive doublewords.

Syntax

Item	Description
.llong	<i>Expression[,Expression,...]</i>

Description

The **.llong** pseudo-op assembles expressions into consecutive doublewords. In 32-bit mode, alignment occurs on fullword boundaries as necessary. In 64-bit mode, alignment occurs on double-word boundaries as necessary.

Parameters

Item	Description
<i>Expression</i>	Represents any expression to be assembled into fullwords/doublewords.

Examples

The following example illustrates the use of the **.llong** pseudo-op:

```
.extern  fooble
.llong   24,3,fooble-333,0
```

which fills 4 doublewords, or 32 bytes, of storage.

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.short pseudo-op” on page 552

“.vbyte pseudo-op” on page 565

“.llong pseudo-op” on page 544

.machine pseudo-op

Purpose

Defines the intended target environment.

Syntax

Item	Description
.machine	<i>StringConstant</i>

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

Item*StringConstant***Description**

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 64-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 or pwrx

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.

pwr4 or 620

Specifies the POWER4 mode. A source program can contain only instructions compatible with the POWER4 processor.

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.

pwr5x Specifies the POWER5+™ mode. A source program can contain only instructions compatible with the POWER5+™ processor.

Item	Description
pwr6	Specifies the POWER6® mode. A source program can contain only instructions compatible with the POWER6® processor.
pwr6e	Specifies the POWER6E mode. A source program can contain only instructions compatible with the POWER6E processor.
pwr7	Specifies the POWER7 mode. A source program can contain only instructions compatible with the POWER7 processor.
pwr8	Specifies the POWER8 mode. A source program can only contain instructions compatible with the POWER8 processor
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.
ppc970 or 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. Any other instruction causes an error.

Examples

- To set the target environment to POWER® family architecture, POWER® family implementation:

```
.machine "pwr"
```
- To set the target environment to any non-specific POWER® family/PowerPC® architecture or implementation mode:

```
.machine "any"
```
- To explicitly select the default assembly mode:

```
.machine ""
```
- The following example of assembler output for a fragment of code shows the usage of `.machine "push"` and `.machine "pop"`:

```

push1.s
File# Line#  Mode Name   Loc Ctr   Object Code      04/15/94
0      1      |          V4.1      Source
0      2      |          .machine  "pwr2"
0      3      |          .csect    longname1[PR]
0      4      |          .long     10
0      5      |          ai       20,30,10
0      6      |          l        10, 20(20)
0      7      |          PWR2 longna 00000000 0000000a
0      8      |          PWR2 longna 00000004 329e000a
0      9      |          PWR2 longna 00000008 81540014

```

0	6					.machine "push"
0	7					.machine "ppc"
0	8					.csect a2[PR]
0	9	PPC	a2	00000000	7d4c42e6	mftb 10
0	10					.machine "pop"
0	11	PWR2	a2	00000004	329e000a	ai 20,30,10
0	12					

Related concepts:

"Pseudo-ops overview" on page 512

A pseudo-op is an instruction to the assembler.

"Assembling with the as command" on page 73

The **as** command invokes the assembler.

"Fixed-point processor" on page 21

The fixed point processor uses non privileged instructions, and GPRs are used as internal storage mechanism.

"Fixed-point rotate and shift instructions" on page 26

The fixed-point rotate and shift instructions rotate the contents of a register.

.org pseudo-op

Purpose

Sets the value of the current location counter.

Syntax

Item	Description
.org	<i>Expression</i>

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

Item	Description
<i>Expression</i>	Represents the value of the current location counter.

Examples

The following example illustrates the use of the **.org** pseudo-op:

```
# Assume assembler location counter is 0x114.
.org $+100
#Skip 100 decimal byte (0x64 bytes).
.
.
# Assembler location counter is now 0x178.
```

Related concepts:

"Pseudo-ops overview" on page 512

A pseudo-op is an instruction to the assembler.

".space pseudo-op" on page 554

.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

Item	Description
<code>.quad</code>	<i>FloatingConstant</i>

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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.float pseudo-op” on page 537

“.double pseudo-op” on page 527

.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 psuedo-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

Item	Description
<i>Name</i>	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]
       l 10, 20(20)
       .long 0xff
       .csect a2[pr]
       .set  r10,10
       .extern C4
C2:    .long 10
C3:    .long 20
       .ref C1,C2,C3
       .ref C4

```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“Combination handling of expressions” on page 63

Terms within an expression can be combined with binary operators.

.rename pseudo-op

Purpose

Creates a synonym or alias for an illegal or undesirable name.

Syntax

Item	Description
.rename	<i>Name, StringConstant</i>

Description

The restrictions on the characters that can be used for symbols within an assembler source file are defined in “Constructing symbols” on page 40. The symbol cannot contain any blanks or special characters, and cannot begin with a digit.

For any global symbol that must contain special characters, or characters that are otherwise illegal in the assembler syntax, the **.rename** pseudo-op provides a way to do so.

The **.rename** pseudo-op changes the *Name* parameter to the *StringConstant* value for all global symbols at the end of assembly. Internal references to the local assembly are made to *Name*. The global name is *StringConstant*.

Parameters

Item	Description
<i>Name</i>	Represents a symbol. To be global, the <i>Name</i> parameter must appear in an .extern or .globl statement.
<i>StringConstant</i>	Represents the value to which the <i>Name</i> 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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“Constructing symbols” on page 40

The Symbols consist of numeric digits, underscores, periods, or lowercase letters.

“.extern pseudo-op” on page 535

“.globl pseudo-op” on page 538

.set pseudo-op

Purpose

Sets a symbol equal to an expression in both type and value.

Syntax

Item	Description
.set	<i>Name</i> , <i>Expression</i>

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

Item	Description
<i>Name</i>	Represents a symbol that may be used before its definition in a .set statement; forward references are allowed within a module.
<i>Expression</i>	Defines the type and the value of the symbol <i>Name</i> . 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 <i>Expression</i> 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.

.
.
lil ap,2

          # Assembler substitutes value 14
          # for the symbol.
          # Note that ap is a register
          # number in context
          # as lil's operand.
```

2. The following example will result in an assembly error because of an invalid type:

```
.csect a1[PR]
L1: 1 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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“Expressions” on page 57

An expression is formed by one or more terms.

.short pseudo-op

Purpose

Assembles expressions into consecutive halfwords.

Syntax

Item	Description
.short	<i>Expression</i> [, <i>Expression</i> ,...]

Description

The **.short** pseudo-op assembles *Expressions* into consecutive halfwords. Halfword alignment occurs as necessary.

Parameters

Item	Description
<i>Expression</i>	Represents expressions that the instruction assembles into halfwords. The <i>Expression</i> parameter cannot refer to the contents of any register. If the <i>Expression</i> 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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.byte pseudo-op” on page 521

“.long pseudo-op” on page 544

“.vbyte pseudo-op” on page 565

.source pseudo-op

Purpose

Identifies the source language type.

Syntax

Item	Description
.source	<i>StringConstant</i>

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

Item	Description
<i>StringConstant</i>	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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“Source language type” on page 5

The assembler records the source language type.

.space pseudo-op

Purpose

Skips a specified number of bytes in the output file and fills them with binary zeros.

Syntax

Item	Description
<i>.space</i>	<i>Number</i>

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

Item	Description
<i>Number</i>	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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

.stabx pseudo-op

Purpose

Provides additional information required by the debugger.

Syntax

Item	Description
.stabx	<i>StringConstant</i> , <i>Expression1</i> , <i>Expression2</i> , <i>Expression3</i>

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

Item	Description
<i>StringConstant</i>	Provides required Stabstring information to the debugger.
<i>Expression1</i>	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).
<i>Expression2</i>	Represents the storage class of the character string.
<i>Expression3</i>	Represents the symbol type of the character string.

Examples

The following example illustrates the use of the **.stabx** pseudo-op:

```
.stabx "INTEGER:t2=-1",0,140,4
```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.function pseudo-op” on page 537

Related information:

Debug Section

.string pseudo-op

Purpose

Assembles character values into consecutive bytes and terminates the string with a null character.

Syntax

Item	Description
<code>.string</code>	<i>StringConstant</i>

Description

The **.string** pseudo-op assembles the character values represented by *StringConstant* into consecutive bytes and terminates the string with a null character.

Parameters

Item	Description
<i>StringConstant</i>	Represents a string of character values assembled into consecutive bytes.

Examples

The following example illustrates the use of the **.string** pseudo-op:

```
mine:  .string "Hello, world!"  
# This produces  
# 0x48656C6C6F2C20776F726C642100.
```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.byte pseudo-op” on page 521

“.vbyte pseudo-op” on page 565

.tbttag pseudo-op

Purpose

Defines a debug traceback tag, preceded by a word of zeros, that can perform tracebacks for debugging programs.

Syntax

.tbttag *Expression1*, *Expression2*, *Expression3*, *Expression4*, *Expression5*, *Expression6*, *Expression7*, *Expression8*, [*Expression9*, *Expression10*, *Expression11*, *Expression12*, *Expression13*, *Expression14*, *Expression15*, *Expression16*]

Description

The **.tbttag** 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

Item	Description	Description
<i>Expression1</i>	version	/*Traceback format version */
Byte	Byte	Byte
<i>Expression2</i>	lang	/*Language values */
Byte	Byte	Byte
	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
<i>Expression3</i>		/*Traceback control bits */
Byte	Byte	Byte
	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.
	tocless	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.
<i>Expression4</i>		/*Traceback control bits (continued) */
Byte	Byte	Byte
	int_hndl	Bit 7. Set if routine is interrupt handler.
	name_present	Bit 6. Set if name is present in proc table.
	uses_alloca	Bit 5. Set if alloca used to allocate storage.
	cl_dis_inv	Bits 4, 3, 2. On-condition directives WALK_ONCOND,0,Walk stack; don't restore state DISCARD_ONCOND,1,Walk the stack and discard. INVOKE_ONCOND,1,Invoke specific system routine
	saves_cr	Bit 1. Set if procedure saves condition register.
	saves_lr	Bit 0. Set if procedure saves link register.
<i>Expression5</i>		/*Traceback control bits (continued) */
Byte	Byte	Byte
	stores_bc	Bit 7. Set if procedure stores the backchain.
	spare2	Bit 6. Spare bit.
	fpr_saved	Bits 5, 4, 3, 2, 1, 0. Number of FPRs saved, max 32.
<i>Expression6</i>		/*Traceback control bits (continued) */
Byte	Byte	Byte
	spare3	Bits 7, 6. Spare bits.
	gpr_saved	Bits 5, 4, 3, 2, 1, 0. Number of GPRs saved, max 32.
<i>Expression7</i>	fixedparms	/*Traceback control bits (continued) */
Byte	Byte	Byte
<i>Expression8</i>		
Byte	Byte	Byte
	floatparms	Bits 7, 6, 5, 4, 3, 2,1. Number of floating point parameters.
	parmsonstk	Bit 0. Set if all parameters placed on stack.

Item	Description	Description
<i>Expression9</i>	parminfo	/*Order and type coding of parameters */
Word	Word	Word
	'0'	Fixed parameter.
	'10'	Single-precision float parameter.
	'11'	Double-precision float parameter.
<i>Expression10</i>	tb_offset	/*Offset from start of code to tb table */
Word	Word	Word
<i>Expression11</i>	hand_mask	/*What interrupts are handled */
Word	Word	Word
<i>Expression12</i>	ctl_info	/*Number of CTL anchors */
Word	Word	Word
<i>Expression13</i>	ctl_info_disp	/*Displacements of each anchor into stack*/
Word	Word	Word
<i>Expression14</i>	name_len	/*Length of procedure name */
Halfword	Halfword	Halfword
<i>Expression15</i>	name	/*Name */
Byte	Byte	Byte
<i>Expression16</i>	alloca_reg	/*Register for alloca automatic storage*/
Byte	Byte	Byte

Examples

The following example illustrates the use of the **.tbttag** pseudo-op:

```
.tbttag 1,0,0xff,0,0,16,0,0
```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.tbttag pseudo-op” on page 556

“.byte pseudo-op” on page 521

.tc pseudo-op

Purpose

Assembles expressions into a Table of Contents (TOC) entry.

Syntax

Item	Description
.tc	[<i>Name</i>][TC], <i>Expression</i> [, <i>Expression</i> ,...]

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

Item	Description
<i>Name</i>	Specifies name of the TOC entry created. The <i>StorageMappingClass</i> is TC for TOC entries. <i>Name</i> [TC] can be used to refer to the TOC entry where appropriate.
<i>Expression</i>	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],proga[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.
```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.csect pseudo-op” on page 525

“.toc pseudo-op”

“.toconf pseudo-op” on page 560

.toc pseudo-op

Purpose

Defines the table of contents of a module.

Syntax

.toc

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]

# A .csect should precede any statements following a .toc/.tc
# section which do not belong in the TOC.

.long TOC[t0]

# The address of the TOC for this module is placed in a fullword.
```

Related concepts:

“.tc pseudo-op” on page 558

“.tocof pseudo-op”

.tocof pseudo-op

Purpose

Allows for the definition of a local symbol to be the table of contents of an external symbol so that the local symbol can be used in expressions.

Syntax

Item	Description
.tocof	<i>Name1, Name2</i>

Description

The **.tocof** pseudo-op declares the *Name2* parameter to be a global symbol and marks the *Name1* symbol as the table of contents (TOCs) 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 the called 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 **.tocof** 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

Item	Description
<i>Name1</i>	Specifies a local symbol that acts as the TOC of a module that contains the <i>Name2</i> value. The <i>Name1</i> symbol should appear in .tc statements.
<i>Name2</i>	Specifies a global symbol that exists within a module that contains a TOC.

Examples

The following example illustrates the use of the **.tocof** pseudo-op:

```

tocbeg:  .toc
apb:    .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.
.tocof tpb,pb
.set always,0x14
.csect [PR]
.using tocbeg,rtoc
l 14,apb
# Load R14 with the address
# of pb.
l 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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“.tc pseudo-op” on page 558

“.toc pseudo-op” on page 559

.using pseudo-op

Purpose

Allows the user to specify a base address and assign a base register number.

Syntax

Item	Description
.using	<i>Expression, Register</i>

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 1 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
d1: .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
      .toc
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
        .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
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
        cal 14, base2(12)     # Load addr. of base2 into r14
        .using base2, 14
        1 4, d12              # Convert to: 1 4, 0xc(14)

```

```

                                # base address base2 is used
1 14, data_block.T(2)          # Load addr. of data[PR] into r14
.using data[PR], 14
1 4, d12                        # Convert to: 1 4, 0x14(14)
.toc
data_block.T: .tc data_block[tc], data[PR]

```

Example 2:

```

.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
ca1 14, base2(12)              # Load addr. of base2 into r14
.using base2, 14
1 4, d12                        # Convert to: 1 4, 0xc(14)
.drop 14
.using data[PR], 12
1 4, d12                        # Convert to: 1 4, 0x14(12)
.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
d1: .long 0x2
base2; .long 0x3
      .long 0x4
      .long 0x5
d12: .long 0x6
1 4, d12                        # 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
.using data[PR], 12
1 4, d12
1 4, 0x14(12)
.drop 12
1 4, d12                        # Error is reported here
.using data[PR], 14
1 4, d12
1 4, 0x14(14)
.toc
data_block.T: .tc data_block[tc], data[PR]
.csect data1[PR]
d13: .long 0x7
      .using data[PR], 5
1 5, d13                        # Error is reported
                                # here, d13 is in csect
                                # data1[PR] and
                                # Using table has no entry of
                                # csect data1[PR]
1 5, d12                        # No error, because d12 is in
                                # data [PR]

```

Parameters

Item	Description
<i>Register</i>	Represents the register number for expressions. It must be absolute and must evaluate to an integer from 0 to 31 inclusive.
<i>Expression</i>	Specifies a label or an expression involving a label that represents the displacement or relative offset into the program. The <i>Expression</i> 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:

```
.csect data[rw]
.long 0x0, 0x0
d1:      .long 0x25
# A read/write csect contains the label d1.
.csect text[pr]
.using data[rw], 12
l 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 concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“Implicit-based addressing” on page 71

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.

“.csect pseudo-op” on page 525

“.drop pseudo-op” on page 528

.vbyte pseudo-op

Purpose

Assembles the value represented by an expression into consecutive bytes.

Syntax

Item	Description
.vbyte	<i>Number, Expression</i>

Description

The **.vbyte** pseudo-op assembles the value represented by the *Expression* parameter into a specified number of consecutive bytes.

Parameters

Item	Description
<i>Number</i>	Specifies a number of consecutive bytes. The <i>Number</i> value must range between 1 and 4.
<i>Expression</i>	Specifies a value that is assembled into consecutive bytes. The <i>Expression</i> parameter cannot contain externally defined symbols. If the <i>Expression</i> 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:

```
.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]
l 3,mine(4)
# GPR 3 now holds 0x37CCFF.
```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.byte pseudo-op” on page 521

.weak pseudo-op

Purpose

Declares a symbol to be a global symbol with weak binding.

Syntax

.weak *Name* [, *Visibility*]

Description

The **.weak** pseudo-op indicates that the symbol *Name* is a global symbol with weak binding, which can be referred to by other files at link time. The **.extern**, **.globl**, or **.comm** pseudo-op can also be used to make a global symbol.

Once the **.weak** pseudo-op has been used for a symbol, using the **.globl**, **.extern**, or **.comm** pseudo-op for the same symbol does not affect the symbol’s weak binding property.

The linker ignores duplicate definitions for symbols with weak binding. If a global symbol is not weak in one file, and is weak in other files, the global definition is used and the weak definitions are ignored. If all definitions are weak, the first weak definition is used.

The visibility of the weak symbol can be specified by using the *Visibility* parameter.

Parameters

Item	Description
<i>Name</i>	Declares <i>Name</i> as a global symbol with weak binding. <i>Name</i> can be a <i>Qualname</i> . (A <i>Qualname</i> specifies a <i>Name</i> and <i>StorageMappingClass</i> for the control section.)
<i>Visibility</i>	Specifies the visibility of the symbol. Valid visibility values are <i>exported</i> , <i>hidden</i> , <i>internal</i> , and <i>protected</i> . Symbol visibilities are used by the linker.

Examples

The following example illustrates the use of the **.weak** pseudo-op:

```
.weak foo[RW]
.csect data[RW]
```

Related concepts:

“Pseudo-ops overview” on page 512

A pseudo-op is an instruction to the assembler.

“.globl pseudo-op” on page 538

“.extern pseudo-op” on page 535

“Visibility of symbols” on page 46

Related information:

ld

.xline pseudo-op

Purpose

Represents a line number.

Syntax

Item	Description
.xline	<i>Number1</i> , <i>StringConstant</i> [, <i>Number2</i>]

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

Item	Description
<i>Number1</i>	Represents the line number of the original source file.
<i>StringConstant</i>	Represents the file name of the original source file.
<i>Number2</i>	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 concepts:

“Pseudo-ops overview” on page 512
A pseudo-op is an instruction to the assembler.

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.

Item	Description
1252-001	<name> is defined already. Cause The user has previously used <i>name</i> 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 <i>expr</i> 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 <i>expr</i> parameter is not defined. Action Verify that all elements of the <i>expr</i> parameter are defined before the .set statement.

Item	Description
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 <i>Name</i> 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. a. Relocatable expressions include label names, .lcomm, names, .comm 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 (.tcof label or name) • unknown (undefined in Pass 2 of the assembler) Action Ensure that the <i>Name</i> 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 <i>Qualname</i> 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 .tcof <i>name1</i> is not valid. Check that the <i>name1</i> has not been defined previously. Cause The <i>Name1</i> parameter of the .tcof pseudo-op has been defined elsewhere in the current module. Action: Ensure that the <i>name1</i> symbol is defined only in the .tcof 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 <i>Name2</i> is not valid. Make sure that <i>name2</i> is an external symbol. Cause The <i>Name2</i> parameter for the .tcof pseudo-op has not been properly defined. Action Ensure that the <i>Name2</i> 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 <i>Name2</i> parameter is defined locally and is externalized using a .extern pseudo-op, this message is also displayed.

Item	Description
1252-011	A .space parameter is undefined. Cause The <i>Number</i> parameter to the .space pseudo-op must be a positive absolute expression. This message indicates that the <i>Number</i> parameter contains an undefined element (such as a label or name for a .lcomm, or .csect pseudo-op that will be defined later). Action Verify that the <i>Number</i> 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 <i>Number</i> parameter to the .space pseudo-op must be a positive absolute expression. This message indicates that the <i>Number</i> parameter contains a nonabsolute element (such as a label or name for a .lcomm, .comm, or .csect pseudo-op). Action Verify that the <i>Number</i> 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 <i>Number</i> parameter to the .space pseudo-op must be a positive absolute expression. This message indicates that the <i>Number</i> parameter resolves to a negative absolute expression. Action Verify that the <i>Number</i> parameter is a positive absolute expression.
1252-014	The .rename <i>Name</i> symbol must be defined in the source code. Cause The <i>Name</i> parameter to the .rename pseudo-op must be defined somewhere in the source code. This message indicates that the <i>Name</i> parameter has not been defined. Action Verify that the <i>Name</i> 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 <i>args</i> 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 <i>Expression</i> 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 <i>Expression</i> parameter of the .byte or .vbyte pseudo-op does not contain externally defined symbols.
1252-020	Do not specify externally defined symbols as .short <i>Expression</i> parameters. Cause If the <i>Expression</i> 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 <i>Expression</i> parameter of the .short pseudo-op does not contain externally defined symbols.
1252-021	The expression must be absolute. Cause The <i>Expression</i> parameter of the .vbyte pseudo-op is not an absolute expression. Action Ensure that the expression is an absolute expression.

Item	Description
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 <i><name></i> 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 <i>Name</i> parameter of a .csect, .comm, .lcomm, .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 <i>Number</i> parameter set to 1 prior to the .short pseudo-op will perform the same function. A .byte pseudo-op with an <i>Expression</i> 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 <i><filename></i>. 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.

Item	Description
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 <i>listfile</i> or the <i>xcrossfile</i> 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 57 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 57 for the specific rules of the valid arithmetic combinations for expressions.

Item	Description
1252-044	The specified source character <i><char></i> 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, <code>.long 3@1</code>, the <code>@</code> 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 <i><filename></i>. 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.
1252-049	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.
1252-050	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.
1252-051	There is an <i>sclass</i> type of <i>Number=<number></i> 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.
1252-052	The specified .align parameter must be an absolute value from 0 to 12. Cause The <i>Number</i> 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 <i>Number</i> parameter resolves into an absolute expression ranging from 0 to 12.
1252-053	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.
2363-054	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.

Item	Description
1252-055	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.
1252-056	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 (<i>Number</i>) 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.

Item	Description
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 <i>name</i> already. Check that this is the only .hash statement associated with the symbol name. Cause The <i>Name</i> parameter of the .hash pseudo-op has already been assigned a string value in a previous .hash statement. Action Ensure that the <i>Name</i> parameter is unique for each .hash pseudo-op.
1252-069	The character <i><char></i> 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 <i>StringConstant</i> parameter are contained within this set.
1252-070	The specified symbol or symbol type for the hash value is not valid. Cause If the <i>Name</i> 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 <i>Name</i> 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.

Item	Description
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 -I 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 <i><positionnumber></i> 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 137 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 1252-086	Obsolete messages. 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.

Item	Description
1252-087	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. Entities and symbol names that are defined as relocatable or non-relocatable are described in “Expressions” on page 57.
1252-088	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).
1252-089 through 1252-098	Obsolete messages.
1252-099	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.
1252-100	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 <i>d(r)</i> operand. Either <i>d</i> or <i>r</i> 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 <i>d</i> or <i>r</i> does not need to be specified, 0 should be put in the place.
1252-101 and 1252-102	Obsolete messages.
1252-103	The specified 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.
1252-104	The <i><parm #></i> parameter must be absolute. Cause The indicated parameter must be absolute (nonrelocatable, nonexternal). Action Refer to the specific instruction article for the instruction syntax.
1252-105	Obsolete message.
1252-106	Not currently used.
1252-107	The parameter <i><parm #></i> 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 132 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 “.csect pseudo-op” on page 525 for information on restrictions. Change the source code, then assemble and link the program again.

Item	Description
1252-108	Warning: The alignment for label <i><name></i> 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.
1252-109	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.
1252-110	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.
1252-111	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 <i>Number</i> 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 <i>Number</i> parameter of the .csect pseudo-op specifies a value greater than previous .csect pseudo-ops that have the same <i>Qualname</i>. • The <i>Number</i> 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.
1252-112	Warning: The <i><inst. format></i> 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
1252-113 and 1252-114	Obsolete messages.
1252-115	The sort failed with status <i><number></i>. 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.
1252-116	There is a system error from <i><name></i>. Check the condition of the system sort command or use local problem reporting procedures. Cause <i>name</i> 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.

Item	Description
1252-117	"Assembler:"
	Cause This line defines a header to the standard error output to indicate that it is an assembly program.
1252-118	"line <number>"
	Cause <i>number</i> 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.
1252-119	".xref"
	Cause This message defines the default suffix extension for the file name of the symbol cross-reference file.
1252-120	".lst"
	Cause This message defines the default suffix extension for the file name of the assembler listing file.
1252-121	"SYMBOL FILE CSECT LINENO"
	Cause This line defines the heading of the symbol cross-reference file.
1252-122 to 1252-123	Define several formats used in the assembler listing file.
1252-124	Obsolete, replaced by 1252-179.
1252-125 to 1252-132	Define the spaces or formats for the assembler listing file.
1252-133 to 1252-134	Define formats for output numbers and names.
1252-135	Defines 8 spaces that are used in the listing file.
1252-136	Defines a format used in the listing file.
1252-137 to 1252-140	Formats for output of a number.
1252-141	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.
1252-142	Syntax 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.
1252-143	The .function <i>Size</i> must be an absolute expression.
	Cause The <i>Size</i> parameter of the .function pseudo-op represents the size of the function. It must be an absolute expression.
	Action Change the <i>Size</i> parameter, then assemble and link the program again.
1252-144	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.
1252-145 and 1252-146	Obsolete, replaced by 1252-180 and 1252-181.
1252-147	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 545 for a list of the defined assembly mode values.
1252-148	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.

Item	Description
1252-149	Instruction <i><name1></i> is not implemented in the current assembly mode <i><name2></i>. 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.
1252-150	The first operand value of <i>value</i> is not valid for PowerPC®. A <i>BO</i> field of 6, 7 14, 15, or greater than 20 is not valid. Cause In branch conditional instructions, the first operand is the <i>B0</i> 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 <i>BO</i> field encoding information to find the correct value of the input operand.
1252-151	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 <i>RA</i> operand not be equal to zero and that it not be equal to <i>RT</i>. 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.
1252-152	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.
1252-153	Warning: Instruction <i><name></i> functions differently between PowerPC® and POWER. Cause This warning message is not displayed unless the -w 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 144 for information on instructions that have the same op code but are functionally different in POWER and PowerPC®.
1252-154	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 <i>L</i> field must be zero for 32-bit implementation. Also, if the mtsri instruction is used in one of the PowerPC® assembly modes, the <i>RA</i> 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.
1252-155	Displacement must be divisible by 4. Cause If an instruction has the <i>DS</i> 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.
1252-156	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 <i>SH</i> field, <i>MB</i> field, or <i>ME</i> 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 137 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.

Item	Description
1252-157	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 137 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.
1252-158	Warning: Special-purpose register number 6 is used to designate the DEC register when the assembly mode is <i>name</i>. 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.
1252-159	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.
1252-160	Warning: A hash code value should be 10 bytes long. Cause When the .hash pseudo-op is used, the second parameter, <i>StringConstant</i>, 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.
1252-161	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 an fwrite, putc, or 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.
1252-162	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 “Assembling and linking a program” on page 73 for the defined assembly modes.
1252-163	The first operand's value <value> is not valid for PowerPC®. The third bit of the B0 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 179, the instruction form is invalid and this message is displayed. Action Change the third bit to one, then assemble and link the program again.
1252-164	This instruction form is not valid for PowerPC®. <i>RA</i>, and <i>RB</i> if present in the instruction, cannot be in the range of registers to be loaded. Also, <i>RA=RT=0</i> is not allowed. Cause In multiple register load instructions, PowerPC® requires that the <i>RA</i> operand, and the <i>RB</i> operand if present in the instruction format, not be in the range of registers to be loaded. Also <i>RA=RT=0</i> is not allowed. Otherwise, this message is displayed. Action Check the register number of the <i>RA</i>, <i>RB</i>, or <i>RT</i> operand to ensure that this requirement is met.
1252-165	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 <i>SV</i> 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.

Item	Description
1252-166	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 <i>RA</i> operand not be equal to zero. Otherwise, this message is displayed. Action Check the register number specified by the <i>RA</i> operand, then assemble and link the source code again.
1252-167	Specify a name with the <code>-<flagname></code> flag. Cause The <code>-n</code> and <code>-o</code> flags of the <code>as</code> command require a filename as a parameter. The <code>-m</code> flag of the <code>as</code> 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 <code>-n</code> and <code>-o</code> flags of the <code>as</code> command, and provide a mode name with the <code>-m</code> flag of the <code>as</code> command.
1252-168	<code>-<name></code> 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.
1252-169	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.
1252-170	The Assembler command has the following syntax: <code>as -l[<i>ListFile</i>] -s[<i>ListFile</i>] -n <i>Name</i> -o <i>ObjectFile</i> [-w -W] -x[<i>XCrossFile</i>] -u -m <i>ModeName</i> [<i>InputFile</i>]</code> Cause This message displays the usage of the <code>as</code> command. Action None.
1252-171	The displacement must be greater than or equal to <code><value1></code> and less than or equal to <code><value2></code>. 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.
1252-172	The <code>.extern</code> symbol is not valid. Check that the <code>.extern <i>Name</i></code> is a relocatable expression. Cause The <i>Name</i> parameter of the <code>.extern</code> pseudo-op must specify a relocatable expression. This message is displayed if the <i>Name</i> parameter of the <code>.extern</code> pseudo-op does not specify a relocatable expression. For information on relocatable and nonrelocatable expressions, see message 1252-004. Action Ensure that the <i>Name</i> parameter of the <code>.extern</code> pseudo-op is a relocatable expression.
1252-173	Warning: The immediate value for instruction <code><name></code> is <code><value></code>. 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 <code>addis</code> instruction (or the <code>lis</code> extended mnemonic of the <code>addis</code> 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 <code>cau</code> 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 <code>addis</code> instruction to construct an unsigned integer. The <code>addis</code> 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 <code>addis</code> 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.
1252-174	Too many <code>.machine "push"</code> instructions without corresponding <code>.machine "pop"</code> instructions. Cause The maximum size of the assembly stack has been exceeded. More than 100 entries have been added to the stack with <code>.machine "push"</code> but not removed with <code>.machine "pop"</code>. Action Change the source program to eliminate the assembly stack overflow condition.

Item	Description
1252-175	A <code>.machine "pop"</code> is seen without a matching <code>.machine "push"</code>. Cause Pseudo-op <code>.machine "pop"</code> attempted to remove an entry from the assembly stack, but the stack is empty. The source program may be missing a <code>.machine "push"</code>. Action Correct the source program.
1252-176	The <code>.ref</code> pseudo-op cannot appear in section <code><name></code>. Cause A <code>.ref</code> 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.
1252-177	The operand of the <code>.ref <name></code> is not a relocatable symbol. Cause <code>.ref</code> pseudo-op operand <code>name</code> 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 <code>.set</code> operand which represents an item that is not relocatable, or a constant value. Action Correct the source program.
1252-178	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.
1252-179	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®.
1252-180	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).
1252-181	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).
1252-182	Storage mapping class <code><name></code> is not valid for <code>.comm</code> pseudo-op. RW is used as the storage mapping class for the object code. Cause The storage mapping class of the <code>.comm</code> 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.
1252-183	TD csect only allowed inside <code>".toc"</code> scope. Cause A csect with storage mapping class TD has been used without first using the <code>.toc</code> pseudo-op. Action Use the <code>.toc</code> pseudo-op before this instruction.
1252-184	TOC anchor must be defined to use a TOC-relative reference to <code><name></code>. Include a <code>.toc</code> 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 <code>.toc</code> pseudo-op in the source program. Action Use the <code>.toc</code> pseudo-op in the program.
1252-185	Warning: Operand is missing from pseudo-op. Cause An operand required for pseudo-ops <code>.byte</code>, <code>.vbyte</code>, <code>.short</code>, <code>.long</code>, or <code>.llong</code> is missing. Action Provide an initial value for the data storage area created by these pseudo-ops.

Item	Description
1252-186	Warning: The maximum length of a stabstring is <i><number></i> 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.
1252-187	Warning: The alignment of the current csect is less than the alignment specified with the <code>.align</code> pseudo-op. Cause The alignment of the csect is not as strict as the alignment required by the use of a <code>.align</code> pseudo-op within that csect. Action The <code>.align</code> 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 <code>.long</code> within the csect requires doubleword alignment, there is a potential for the <code>.long</code> value to ultimately (after linking) be only word-aligned. This may not be what is intended by the user.
1252-188	Zero is used in the L operand for the <i><instruction></i> 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.
1252-189	Invalid value for environment variable <code>OBJECT_MODE</code>. Set the <code>OBJECT_MODE</code> environment variable to 32 or 64 or use the <code>-a32</code> or <code>-a64</code> option. Cause The value of the <code>OBJECT_MODE</code> environment variable is not recognized by the assembler. Action Set the <code>OBJECT_MODE</code> environment variable to either 32 or 64, or use the <code>-a32</code> or <code>-a64</code> command line option. Any other value for the environment variable has no meaning to the assembler.
1252-190	Invalid reference to label <i><name></i>: <code>.function</code> pseudo-op must refer to a csect. Cause The <code>.function</code> pseudo-op referred to a local label. Action The reference <i><name></i> should be the name (label) of a csect.
1252-191	Only <i><name></i> should be used for relocatable expressions. Cause The expression used to initialize <i><name></i> contains references to externally defined symbols (i.e. the symbols appear in <code>.extern</code> pseudo-op). Action Verify that no externally defined symbols are contained within the expression operands for <i><name></i>. 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.
1252-192	Assembly mode is not specified. Set the <code>OBJECT_MODE</code> environment variable to 32 or 64 or use the <code>-a32</code> or <code>-a64</code> option. Cause The environment variable contains the value <code>32_64</code>. Action Set the <code>OBJECT_MODE</code> environment variable to either 32 or 64, or use the <code>-a32</code> or <code>-a64</code> command line option.
1252-193	Values specified with the <code>.set</code> pseudo-op are treated as 32-bit signed numbers. Unexpected results may occur when these values are used in a <code>.llong</code> expression. Cause In 32-bit mode, an expression that results from the use of <code>.set</code> has been used to set the initial value of a <code>.llong</code>. Action For initializing <code>.llong</code>'s when in 32-bit mode, values are treated as 64-bit. If a <code>.set</code> symbol whose most significant bit 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 <code>0xFFFF_0000</code> 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 <code>0xFFFF_FFFF_FFFF_0000</code>.
1252-194	Warning: The immediate value for instruction <i><instruction></i> is <i><number></i>. It may not be portable to a 64-bit machine if this value is to be treated as an unsigned value. Cause This is an 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:

Implementation	Description
com	Supported by POWER family, POWER2™, and PowerPC implementations.
POWER family	Supported only by POWER family and POWER2™ implementations.
POWER2™	Supported only by POWER2™ implementations.
PowerPC	Supported only by PowerPC architecture.
PPC opt.	Defined only in PowerPC architecture and is an optional instruction.
603 only	Supported only on the PowerPC 603 RISC Microprocessor

Item	Description	Implementation	Format	Primary Op Code	Extended Op Code
Instruction Set Sorted by Mnemonic	Instruction Set Sorted by Mnemonic	Instruction Set Sorted by Mnemonic	Instruction Set Sorted by Mnemonic	Instruction Set Sorted by Mnemonic	Instruction Set Sorted by Mnemonic
Mnemonic	Instruction	Implementation	Format	Primary Op Code	Extended Op Code
a[o][.]	Add Carrying	POWER family	XO	31	10
abs[o][.]	Absolute	POWER family	XO	31	360
add[o][.]	Add	PowerPC	XO	31	266
addc[o][.]	Add Carrying	PowerPC	XO	31	10
adde[o][.]	Add Extended	PowerPC	XO	31	138
addi	Add Immediate	PowerPC	D	14	
addic	Add Immediate Carrying	PowerPC	D	12	
addic.	Add Immediate Carrying and Record	PowerPC	D	13	
addis	Add Immediate Shifted	PowerPC	D	15	
addme[o][.]	Add to Minus One Extended	PowerPC	XO	31	234
addze[o][.]	Add to Zero Extended	PowerPC	XO	31	202
ae[o][.]	Add Extended	POWER family	XO	31	138
ai	Add Immediate	POWER family	D	12	
ai.	Add Immediate and Record	POWER family	D	13	
ame[o][.]	Add to Minus One Extended	POWER family	XO	31	234
and[.]	AND	com	X	31	28
andc[.]	AND with Complement	com	X	31	60
andi.	AND Immediate	PowerPC	D	28	
andil.	AND Immediate Lower	POWER family	D	28	
andis.	AND Immediate Shifted	PowerPC	D	29	
andiu.	AND Immediate Upper	POWER family	D	29	
aze[o][.]	Add to Zero Extended	POWER family	XO	31	202

Item	Description	Implementation	Format	Primary Op Code	Extended Op Code
b[l][a]	Branch	com	I	18	
bc[l][a]	Branch Conditional	com	B	16	
bcc[l]	Branch Conditional to Count Register	POWER family	XL	19	528
bcctr[l]	Branch Conditional to Count Register	PowerPC	XL	19	528
bclr[l]	Branch Conditional Link Register	PowerPC	XL	19	16
bcr[l]	Branch Conditional Register	POWER family	XL	19	16
cal	Compute Address Lower	POWER family	D	14	
cau	Compute Address Upper	POWER family	D	15	
cax[o][.]	Compute Address	POWER family	XO	31	266
clcs	Cache Line Compute Size	POWER family	X	31	531
clf	Cache Line Flush	POWER family	X	31	118
cli	Cache Line Invalidate	POWER family	X	31	502
cmp	Compare	com	X	31	0
cmpi	Compare Immediate	com	D	11	
cmpl	Compare Logical	com	X	31	32
cmpli	Compare Logical Immediate	com	D	10	
cntlz[.]	Count Leading Zeros	POWER family	X	31	26
cntlzw[.]	Count Leading Zeros Word	PowerPC	X	31	26
crand	Condition Register AND	com	XL	19	257
crandc	Condition Register AND with Complement	com	XL	19	129
creqv	Condition Register Equivalent	com	XL	19	289
crnand	Condition Register NAND	com	XL	19	225
crnor	Condition Register NOR	com	XL	19	33
cror	Condition Register OR	com	XL	19	449
crorc	Condition Register OR with Complement	com	XL	19	417
crxor	Condition Register XOR	com	XL	19	193
dcbf	Data Cache Block Flush	PowerPC	X	31	86
dcbi	Data Cache Block Invalidate	PowerPC	X	31	470
dcbst	Data Cache Block Store	PowerPC	X	31	54

Item	Description	Implementation	Format	Primary Op Code	Extended Op Code
dcbt	Data Cache Block Touch	PowerPC	X	31	278
dcbtst	Data Cache Block Touch for Store	PowerPC	X	31	246
dcbz	Data Cache Block Set to Zero	PowerPC	X	31	1014
dclst	Data Cache Line Store	POWER family	X	31	630
dclz	Data Cache Line Set to Zero	POWER family	X	31	1014
dcs	Data Cache Synchronize	POWER family	X	31	598
div[o][.]	Divide	POWER family	XO	31	331
divs[o][.]	Divide Short	POWER family	XO	31	363
divw[o][.]	Divide Word	PowerPC	XO	31	491
divwu[o][.]	Divide Word Unsigned	PowerPC	XO	31	459
doz[o][.]	Difference or Zero	POWER family	XO	31	264
dozi	Difference or Zero Immediate	POWER family	D	09	
eciwx	External Control in Word Indexed	PPC opt.	X	31	310
ecowx	External Control out Word Indexed	PPC opt.	X	31	438
eieio	Enforce In-order Execution of I/O	PowerPC	X	31	854
eqv[.]	Equivalent	com	X	31	284
exts[.]	Extend Sign	POWER family	X	31	922
extsb[.]	Extend Sign Byte	PowerPC	X	31	954
extsh[.]	Extend Sign Halfword	PowerPC	XO	31	922
fa[.]	Floating Add	POWER family	A	63	21
fabs[.]	Floating Absolute Value	com	X	63	264
fadd[.]	Floating Add	PowerPC	A	63	21
fadds[.]	Floating Add Single	PowerPC	A	59	21
fcir[.]	Floating Convert to Integer Word	POWER family	X	63	14
fcirz[.]	Floating Convert to Integer Word with Round to Zero	POWER family	X	63	15
fcmpo	Floating Compare Ordered	com	X	63	32
fcmpu	Floating Compare Unordered	com	XL	63	0
fctiw[.]	Floating Convert to Integer Word	PowerPC	X	63	14
fctiwz[.]	Floating Convert to Integer Word with Round to Zero	PowerPC	XL	63	15
fd[.]	Floating Divide	POWER family	A	63	18
fddiv[.]	Floating Divide	PowerPC	A	63	18

Item	Description	Implementation	Format	Primary Op Code	Extended Op Code
fdivs[.]	Floating Divide Single	PowerPC	A	59	18
fm[.]	Floating Multiply	POWER family	A	63	25
fma[.]	Floating Multiply-Add	POWER family	A	63	29
fmadd[.]	Floating Multiply-Add	PowerPC	A	63	29
fmadds[.]	Floating Multiply-Add Single	PowerPC	A	59	29
fmr[.]	Floating Move Register	com	X	63	72
fms[.]	Floating Multiply-Subtract	POWER family	A	63	28
fmsub[.]	Floating Multiply-Subtract	PowerPC	A	63	28
fmsubs[.]	Floating Multiply-Subtract Single	PowerPC	A	59	28
fmul[.]	Floating Multiply	PowerPC	A	63	25
fmuls[.]	Floating Multiply Single	PowerPC	A	59	25
fnabs[.]	Floating Negative Absolute Value	com	X	63	136
fneg[.]	Floating Negate	com	X	63	40
fnma[.]	Floating Negative Multiply-Add	POWER family	A	63	31
fnmadd[.]	Floating Negative Multiply-Add	PowerPC	A	63	31
fnmadds[.]	Floating Negative Multiply-Add Single	PowerPC	A	59	31
fnms[.]	Floating Negative Multiply-Subtract	POWER family	A	63	30
fnmsub[.]	Floating Negative Multiply-Subtract	PowerPC	A	63	30
fnmsubs[.]	Floating Negative Multiply-Subtract Single	PowerPC	A	59	30
fres[.]	Floating Reciprocal Estimate Single	PPC opt.	A	59	24
frsp[.]	Floating Round to Single Precision	com	X	63	12
frsqre[.]	Floating Reciprocal Square Root Estimate	PPC opt.	A	63	26
fs[.]	Floating Subtract	POWER family	A	63	20
fsel[.]	Floating-Point Select	PPC opt.	A	63	23
fsqrt[.]	Floating Square Root	POWER2™	A	63	22
fsub[.]	Floating Subtract	PowerPC	A	63	20
fsubs[.]	Floating Subtract Single	PowerPC	A	59	20

Item	Description	Implementation	Format	Primary Op Code	Extended Op Code
icbi	Instruction Cache Block Invalidate	PowerPC	X	31	982
ics	Instruction Cache Synchronize	POWER family	X	19	150
isync	Instruction Synchronize	PowerPC	X	19	150
l	Load	POWER family	D	32	
lbrx	Load Byte-Reversed Indexed	POWER family	X	31	534
lbz	Load Byte and Zero	com	D	34	
lbzu	Load Byte and Zero with Update	com	D	35	
lbzux	Load Byte and Zero with Update Indexed	com	X	31	119
lbzx	Load Byte and Zero Indexed	com	X	31	87
lfd	Load Floating-Point Double	com	D	50	
lfdi	Load Floating-Point Double with Update	com	D	51	
lfdix	Load Floating-Point Double with Update Indexed	com	X	31	631
lfdx	Load Floating-Point Double Indexed	com	X	31	599
lfq	Load Floating-Point Quad	POWER2™	D	56	
lfqu	Load Floating-Point Quad with Update	POWER2™	D	57	
lfqux	Load Floating-Point Quad with Update Indexed	POWER2™	X	31	823
lfqx	Load Floating-Point Quad Indexed	POWER2™	X	31	791
lfs	Load Floating-Point Single	com	D	48	
lfsu	Load Floating-Point Single with Update	com	D	49	
lfsux	Load Floating-Point Single with Update Indexed	com	X	31	567
lfsx	Load Floating-Point Single Indexed	com	X	31	535
lha	Load Half Algebraic	com	D	42	
lhau	Load Half Algebraic with Update	com	D	43	
lhaux	Load Half Algebraic with Update Indexed	com	X	31	375
lhax	Load Half Algebraic Indexed	com	X	31	343
lhbrx	Load Half Byte-Reversed Indexed	com	X	31	790

Item	Description	Implementation	Format	Primary Op Code	Extended Op Code
lhz	Load Half and Zero	com	D	40	
lhzu	Load Half and Zero with Update	com	D	41	
lhzux	Load Half and Zero with Update Indexed	com	X	31	331
lhzx	Load Half and Zero Indexed	com	X	31	279
lm	Load Multiple	POWER family	D	46	
lmw	Load Multiple Word	PowerPC	D	46	
lscbx	Load String and Compare Byte Indexed	POWER family	X	31	277
lsi	Load String Immediate	POWER family	X	31	597
lswi	Load String Word Immediate	PowerPC	X	31	597
lswx	Load String Word Indexed	PowerPC	X	31	533
lsx	Load String Indexed	POWER family	X	31	533
lu	Load with Update	POWER family	D	33	
lux	Load with Update Indexed	POWER family	X	31	55
lwarx	Load Word and Reserve Indexed	PowerPC	X	31	20
lwbrx	Load Word Byte-Reversed Indexed	PowerPC	X	31	534
lwz	Load Word and Zero	PowerPC	D	32	
lwzu	Load Word with Zero Update	PowerPC	D	33	
lwzux	Load Word and Zero with Update Indexed	PowerPC	X	31	55
lwzx	Load Word and Zero Indexed	PowerPC	X	31	23
lx	Load Indexed	POWER family	X	31	23
maskg[.]	Mask Generate	POWER family	X	31	29
maskir[.]	Mask Insert from Register	POWER family	X	31	541
mcrf	Move Condition Register Field	com	XL	19	0
mcrfs	Move to Condition Register from FPSCR	com	X	63	64
mcrxr	Move to Condition Register from XER	com	X	31	512
mfcrr	Move from Condition Register	com	X	31	19
mffs[.]	Move from FPSCR	com	X	63	583
mfmsr	Move from Machine State Register	com	X	31	83

Item	Description	Implementation	Format	Primary Op Code	Extended Op Code
mfspr	Move from Special-Purpose Register	com	X	31	339
mfsr	Move from Segment Register	com	X	31	595
mfsri	Move from Segment Register Indirect	POWER family	X	31	627
mfsrin	Move from Segment Register Indirect	PowerPC	X	31	659
mtrcf	Move to Condition Register Fields	com	XFX	31	144
mtfsb0[.]	Move to FPSCR Bit 0	com	X	63	70
mtfsb1[.]	Move to FPSCR Bit 1	com	X	63	38
mtfsf[.]	Move to FPSCR Fields	com	XFL	63	711
mtfsfi[.]	Move to FPSCR Field Immediate	com	X	63	134
mtmsr	Move to Machine State Register	com	X	31	146
mtspr	Move to Special-Purpose Register	com	X	31	467
mtsr	Move to Segment Register	com	X	31	210
mtsri	Move to Segment Register Indirect	POWER family	X	31	242
mtsrin	Move to Segment Register Indirect	PowerPC	X	31	242
mul[o][.]	Multiply	POWER family	XO	31	107
mulhw[.]	Multiply High Word	PowerPC	XO	31	75
mulhwu[.]	Multiply High Word Unsigned	PowerPC	XO	31	11
muli	Multiply Immediate	POWER family	D	07	
mulli	Multiply Low Immediate	PowerPC	D	07	
mullw[o][.]	Multiply Low Word	PowerPC	XO	31	235
muls[o][.]	Multiply Short	POWER family	XO	31	235
nabs[o][.]	Negative Absolute	POWER family	XO	31	488
nand[.]	NAND	com	X	31	476
neg[o][.]	Negate	com	XO	31	104
nor[.]	NOR	com	X	31	124
or[.]	OR	com	X	31	444
orc[.]	OR with Complement	com	X	31	412
ori	OR Immediate	PowerPC	D	24	
oril	OR Immediate Lower	POWER family	D	24	
oris	OR Immediate Shifted	PowerPC	D	25	

Item	Description	Implementation	Format	Primary Op Code	Extended Op Code
oriu	OR Immediate Upper	POWER family	D	25	
rac[.]	Real Address Compute	POWER family	X	31	818
rfi	Return from Interrupt	com	X	19	50
rfsvc	Return from SVC	POWER family	X	19	82
rlimi[.]	Rotate Left Immediate then Mask Insert	POWER family	M	20	
rlinm[.]	Rotate Left Immediate then AND with Mask	POWER family	M	21	
rlmi[.]	Rotate Left then Mask Insert	POWER family	M	22	
rlnm[.]	Rotate Left then AND with Mask	POWER family	M	23	
rlwimi[.]	Rotate Left Word Immediate then Mask Insert	PowerPC	M	20	
rlwinm[.]	Rotate Left Word Immediate then AND with Mask	PowerPC	M	21	
rlwnm[.]	Rotate Left Word then AND with Mask	PowerPC	M	23	
rrib[.]	Rotate Right and Insert Bit	POWER family	X	31	537
sc	System Call	PowerPC	SC	17	
sf[o][.]	Subtract from	POWER family	XO	31	08
sfe[o][.]	Subtract from Extended	POWER family	XO	31	136
sfi	Subtract from Immediate	POWER family	D	08	
sfme[o][.]	Subtract from Minus One Extended	POWER family	XO	31	232
sfze[o][.]	Subtract from Zero Extended	POWER family	XO	31	200
si	Subtract Immediate	com	D	12	
si.	Subtract Immediate and Record	com	D	13	
sl[.]	Shift Left	POWER family	X	31	24
sle[.]	Shift Left Extended	POWER family	X	31	153
sleq[.]	Shift Left Extended with MQ	POWER family	X	31	217
sliq[.]	Shift Left Immediate with MQ	POWER family	X	31	184
slliq[.]	Shift Left Long Immediate with MQ	POWER family	X	31	248
sllq[.]	Shift Left Long with MQ	POWER family	X	31	216
slq[.]	Shift Left with MQ	POWER family	X	31	152
slw[.]	Shift Left Word	PowerPC	X	31	24

Item	Description	Implementation	Format	Primary Op Code	Extended Op Code
sr[.]	Shift Right	POWER family	X	31	536
sra[.]	Shift Right Algebraic	POWER family	X	31	792
srai[.]	Shift Right Algebraic Immediate	POWER family	X	31	824
sraiq[.]	Shift Right Algebraic, Immediate with MQ	POWER family	X	31	952
sraq[.]	Shift Right Algebraic with MQ	POWER family	X	31	920
sraw[.]	Shift Right Algebraic Word	PowerPC	X	31	792
srawi[.]	Shift Right Algebraic Word Immediate	PowerPC	X	31	824
sre[.]	Shift Right Extended	POWER family	X	31	665
srea[.]	Shift Right Extended Algebraic	POWER family	X	31	921
sreq[.]	Shift Right Extended with MQ	POWER family	X	31	729
sriq[.]	Shift Right Immediate with MQ	POWER family	X	31	696
srlq[.]	Shift Right Long Immediate with MQ	POWER family	X	31	760
srlq[.]	Shift Right Long with MQ	POWER family	X	31	728
srq[.]	Shift Right with MQ	POWER family	X	31	664
srw[.]	Shift Right Word	PowerPC	X	31	536
st	Store	POWER family	D	36	
stb	Store Byte	com	D	38	
stbrx	Store Byte-Reversed Indexed	POWER family	X	31	662
stbu	Store Byte with Update	com	D	39	
stbux	Store Byte with Update Indexed	com	X	31	247
stbx	Store Byte Indexed	com	X	31	215
stfd	Store Floating-Point Double	com	D	54	
stfdu	Store Floating-Point Double with Update	com	D	55	
stfdux	Store Floating-Point Double with Update Indexed	com	X	31	759
stfdx	Store Floating-Point Double Indexed	com	X	31	727
stfiwx	Store Floating-Point as Integer Word Indexed	PPC opt.	X	31	983
stfq	Store Floating-Point Quad	POWER2™	DS	60	

Item	Description	Implementation	Format	Primary Op Code	Extended Op Code
stfqu	Store Floating-Point Quad with Update	POWER2™	DS	61	
stfqux	Store Floating-Point Quad with Update Indexed	POWER2™	X	31	951
stfqx	Store Floating-Point Quad Indexed	POWER2™	X	31	919
stfs	Store Floating-Point Single	com	D	52	
stfsu	Store Floating-Point Single with Update	com	D	53	
stfsux	Store Floating-Point Single with Update Indexed	com	X	31	695
stfsx	Store Floating-Point Single Indexed	com	X	31	663
sth	Store Half	com	D	44	
sthbrx	Store Half Byte-Reverse Indexed	com	X	31	918
sthu	Store Half with Update	com	D	45	
sthux	Store Half with Update Indexed	com	X	31	439
sthx	Store Half Indexed	com	X	31	407
stm	Store Multiple	POWER family	D	47	
stmw	Store Multiple Word	PowerPC	D	47	
stsi	Store String Immediate	POWER family	X	31	725
stswi	Store String Word Immediate	PowerPC	X	31	725
stswx	Store String Word Indexed	PowerPC	X	31	661
stsx	Store String Indexed	POWER family	X	31	661
stu	Store with Update	POWER family	D	37	
stux	Store with Update Indexed	POWER family	X	31	183
stw	Store	PowerPC	D	36	
stwbrx	Store Word Byte-Reversed Indexed	PowerPC	X	31	662
stwcx.	Store Word Conditional Indexed	PowerPC	X	31	150
stwu	Store Word with Update	PowerPC	D	37	
stwux	Store Word with Update Indexed	PowerPC	X	31	183
stwx	Store Word Indexed	PowerPC	X	31	151
stx	Store Indexed	POWER family	X	31	151
subf[o][.]	Subtract from	PowerPC	XO	31	40
subfc[o][.]	Subtract from Carrying	PowerPC	XO	31	08

Item	Description	Implementation	Format	Primary Op Code	Extended Op Code
subfe[o][.]	Subtract from Extended	PowerPC	XO	31	136
subfic	Subtract from Immediate Carrying	PowerPC	D	08	
subfme[o][.]	Subtract from Minus One Extended	PowerPC	XO	31	232
subfze[o][.]	Subtract from Zero Extended	PowerPC	XO	31	200
svc[l][a]	Supervisor Call	POWER family	SC	17	
sync	Synchronize	PowerPC	X	31	598
t	Trap	POWER family	X	31	04
ti	Trap Immediate	POWER family	D	03	
tlbi	Translation Look-aside Buffer Invalidate Entry	POWER family	X	31	306
tlbie	Translation Look-aside Buffer Invalidate Entry	PPC opt.	X	31	306
tlbld	Load Data TLB Entry	603 only	X	31	978
tlbli	Load Instruction TLB Entry	603 only	X	31	1010
tlbsync	Translation Look-aside Buffer Synchronize	PPC opt.	X	31	566
tw	Trap Word	PowerPC	X	31	04
twi	Trap Word Immediate	PowerPC	D	03	
xor[.]	XOR	com	X	31	316
xori	XOR Immediate	PowerPC	D	26	
xoril	XOR Immediate Lower	POWER family	D	26	
xoris	XOR Immediate Shift	PowerPC	D	27	
xoriu	XOR Immediate Upper	POWER family	D	27	

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 39. Instruction Set Sorted by Primary and Extended Op Code

Implementation	Description
com	Supported by POWER family, POWER2™, and PowerPC implementations.
POWER family	Supported only by POWER family and POWER2™ implementations.
POWER2™	Supported only by POWER2™ implementations.
PowerPC	Supported only by PowerPC architecture.
PPC opt.	Defined only in PowerPC architecture and is an optional instruction.
603 only	Supported only on the PowerPC 603 RISC Microprocessor

Item	Description	Description	Description	Description	Description
Instruction Set Sorted by Primary and Extended Op Code	Instruction Set Sorted by Primary and Extended Op Code	Instruction Set Sorted by Primary and Extended Op Code	Instruction Set Sorted by Primary and Extended Op Code	Instruction Set Sorted by Primary and Extended Op Code	Instruction Set Sorted by Primary and Extended Op Code
Mnemonic	Instruction	Implementation	Format	Primary Op Code	Extended Op Code
ti	Trap Immediate	POWER family	D	03	
twi	Trap Word Immediate	PowerPC	D	03	
muli	Multiply Immediate	POWER family	D	07	
mulli	Multiply Low Immediate	PowerPC	D	07	
sfi	Subtract from Immediate	POWER family	D	08	
subfic	Subtract from Immediate Carrying	PowerPC	D	08	
dozi	Difference or Zero Immediate	POWER family	D	09	
cmpli	Compare Logical Immediate	com	D	10	
cmpi	Compare Immediate	com	D	11	
addic	Add Immediate Carrying	PowerPC	D	12	
ai	Add Immediate	POWER family	D	12	
si	Subtract Immediate	com	D	12	
addic.	Add Immediate Carrying and Record	PowerPC	D	13	
si.	Subtract Immediate and Record	com	D	13	
ai.	Add Immediate and Record	POWER family	D	13	
addi	Add Immediate	PowerPC	D	14	
cal	Compute Address Lower	POWER family	D	14	
addis	Add Immediate Shifted	PowerPC	D	15	
cau	Compute Address Upper	POWER family	D	15	
bc[l][a]	Branch Conditional	com	B	16	

Item	Description	Description	Description	Description	Description
sc	System Call	PowerPC	SC	17	
svc[l][a]	Supervisor Call	POWER family	SC	17	
b[l][a]	Branch	com	I	18	
mcrf	Move Condition Register Field	com	XL	19	0
bclr[l]	Branch Conditional Link Register	PowerPC	XL	19	16
bcr[l]	Branch Conditional Register	POWER family	XL	19	16
crnor	Condition Register NOR	com	XL	19	33
rfi	Return from Interrupt	com	X	19	50
rfsvc	Return from SVC	POWER family	X	19	82
crandc	Condition Register AND with Complement	com	XL	19	129
ics	Instruction Cache Synchronize	POWER family	X	19	150
isync	Instruction Synchronize	PowerPC	X	19	150
crxor	Condition Register XOR	com	XL	19	193
crnand	Condition Register NAND	com	XL	19	225
crand	Condition Register AND	com	XL	19	257
creqv	Condition Register Equivalent	com	XL	19	289
crorc	Condition Register OR with Complement	com	XL	19	417
cror	Condition Register OR	com	XL	19	449
bcc[l]	Branch Conditional to Count Register	POWER family	XL	19	528
bcctr[l]	Branch Conditional to Count Register	PowerPC	XL	19	528
rlimi[.]	Rotate Left Immediate then Mask Insert	POWER family	M	20	
rlwimi[.]	Rotate Left Word Immediate then Mask Insert	PowerPC	M	20	
rlinm[.]	Rotate Left Immediate then AND with Mask	POWER family	M	21	
rlwinm[.]	Rotate Left Word Immediate then AND with Mask	PowerPC	M	21	
rlmi[.]	Rotate Left then Mask Insert	POWER family	M	22	
rlnm[.]	Rotate Left then AND with Mask	POWER family	M	23	

Item	Description	Description	Description	Description	Description
rlwnm[.]	Rotate Left Word then AND with Mask	PowerPC	M	23	
ori	OR Immediate	PowerPC	D	24	
oril	OR Immediate Lower	POWER family	D	24	
oris	OR Immediate Shifted	PowerPC	D	25	
oriu	OR Immediate Upper	POWER family	D	25	
xori	XOR Immediate	PowerPC	D	26	
xoril	XOR Immediate Lower	POWER family	D	26	
xoris	XOR Immediate Shift	PowerPC	D	27	
xoriu	XOR Immediate Upper	POWER family	D	27	
andi.	AND Immediate	PowerPC	D	28	
andil.	AND Immediate Lower	POWER family	D	28	
andis.	AND Immediate Shifted	PowerPC	D	29	
andiu.	AND Immediate Upper	POWER family	D	29	
cmp	Compare	com	X	31	0
t	Trap	POWER family	X	31	04
tw	Trap Word	PowerPC	X	31	04
sf[o][.]	Subtract from	POWER family	XO	31	08
subfc[o][.]	Subtract from Carrying	PowerPC	XO	31	08
a[o][.]	Add Carrying	POWER family	XO	31	10
addc[o][.]	Add Carrying	PowerPC	XO	31	10
mulhwu[.]	Multiply High Word Unsigned	PowerPC	XO	31	11
mfcr	Move from Condition Register	com	X	31	19
lwarx	Load Word and Reserve Indexed	PowerPC	X	31	20
lwzx	Load Word and Zero Indexed	PowerPC	X	31	23
lx	Load Indexed	POWER family	X	31	23
sl[.]	Shift Left	POWER family	X	31	24
slw[.]	Shift Left Word	PowerPC	X	31	24
cntlz[.]	Count Leading Zeros	POWER family	X	31	26
cntlzw[.]	Count Leading Zeros Word	PowerPC	X	31	26
and[.]	AND	com	X	31	28
maskg[.]	Mask Generate	POWER family	X	31	29
cmpl	Compare Logical	com	X	31	32
subf[o][.]	Subtract from	PowerPC	XO	31	40

Item	Description	Description	Description	Description	Description
dcbst	Data Cache Block Store	PowerPC	X	31	54
lux	Load with Update Indexed	POWER family	X	31	55
lwzux	Load Word and Zero with Update Indexed	PowerPC	X	31	55
andc[.]	AND with Complement	com	X	31	60
mulhw[.]	Multiply High Word	PowerPC	XO	31	75
mfmsr	Move from Machine State Register	com	X	31	83
dcbf	Data Cache Block Flush	PowerPC	X	31	86
lbzx	Load Byte and Zero Indexed	com	X	31	87
neg[o][.]	Negate	com	XO	31	104
mul[o][.]	Multiply	POWER family	XO	31	107
clf	Cache Line Flush	POWER family	X	31	118
lbzux	Load Byte and Zero with Update Indexed	com	X	31	119
nor[.]	NOR	com	X	31	124
sfe[o][.]	Subtract from Extended	POWER family	XO	31	136
subfe[o][.]	Subtract from Extended	PowerPC	XO	31	136
adde[o][.]	Add Extended	PowerPC	XO	31	138
ae[o][.]	Add Extended	POWER family	XO	31	138
mtcrf	Move to Condition Register Fields	com	XFX	31	144
mtmsr	Move to Machine State Register	com	X	31	146
stwcx.	Store Word Conditional Indexed	PowerPC	X	31	150
stwx	Store Word Indexed	PowerPC	X	31	151
stx	Store Indexed	POWER family	X	31	151
slq[.]	Shift Left with MQ	POWER family	X	31	152
sle[.]	Shift Left Extended	POWER family	X	31	153
stux	Store with Update Indexed	POWER family	X	31	183
stwux	Store Word with Update Indexed	PowerPC	X	31	183
sliq[.]	Shift Left Immediate with MQ	POWER family	X	31	184
sfze[o][.]	Subtract from Zero Extended	POWER family	XO	31	200
subfze[o][.]	Subtract from Zero Extended	PowerPC	XO	31	200
addze[o][.]	Add to Zero Extended	PowerPC	XO	31	202

Item	Description	Description	Description	Description	Description
aze[o][.]	Add to Zero Extended	POWER family	XO	31	202
mtsr	Move to Segment Register	com	X	31	210
stbu	Store Byte with Update	com	D	39	
stbx	Store Byte Indexed	com	X	31	215
sllq[.]	Shift Left Long with MQ	POWER family	X	31	216
sleq[.]	Shift Left Extended with MQ	POWER family	X	31	217
sfme[o][.]	Subtract from Minus One Extended	POWER family	XO	31	232
subfme[o][.]	Subtract from Minus One Extended	PowerPC	XO	31	232
addme[o][.]	Add to Minus One Extended	PowerPC	XO	31	234
ame[o][.]	Add to Minus One Extended	POWER family	XO	31	234
mullw[o][.]	Multiply Low Word	PowerPC	XO	31	235
muls[o][.]	Multiply Short	POWER family	XO	31	235
mtsri	Move to Segment Register Indirect	POWER family	X	31	242
mtsrin	Move to Segment Register Indirect	PowerPC	X	31	242
dcbtst	Data Cache Block Touch for Store	PowerPC	X	31	246
stbux	Store Byte with Update Indexed	com	X	31	247
slliq[.]	Shift Left Long Immediate with MQ	POWER family	X	31	248
doz[o][.]	Difference or Zero	POWER family	XO	31	264
add[o][.]	Add	PowerPC	XO	31	266
cax[o][.]	Compute Address	POWER family	XO	31	266
lscbx	Load String and Compare Byte Indexed	POWER family	X	31	277
dcbt	Data Cache Block Touch	PowerPC	X	31	278
lhzx	Load Half and Zero Indexed	com	X	31	279
eqv[.]	Equivalent	com	X	31	284
tlbi	Translation Look-aside Buffer Invalidate Entry	POWER family	X	31	306
tlbie	Translation Look-aside Buffer Invalidate Entry	PPC opt.	X	31	306
eciwx	External Control in Word Indexed	PPC opt.	X	31	310
xor[.]	XOR	com	X	31	316
div[o][.]	Divide	POWER family	XO	31	331

Item	Description	Description	Description	Description	Description
lhzux	Load Half and Zero with Update Indexed	com	X	31	331
mfspr	Move from Special-Purpose Register	com	X	31	339
lhax	Load Half Algebraic Indexed	com	X	31	343
abs[o][.]	Absolute	POWER family	XO	31	360
divs[o][.]	Divide Short	POWER family	XO	31	363
lhaux	Load Half Algebraic with Update Indexed	com	X	31	375
sthx	Store Half Indexed	com	X	31	407
orc[.]	OR with Complement	com	X	31	412
ecowx	External Control out Word Indexed	PPC opt.	X	31	438
sthux	Store Half with Update Indexed	com	X	31	439
or[.]	OR	com	X	31	444
divwu[o][.]	Divide Word Unsigned	PowerPC	XO	31	459
mtspr	Move to Special-Purpose Register	com	X	31	467
dcbi	Data Cache Block Invalidate	PowerPC	X	31	470
nand[.]	NAND	com	X	31	476
nabs[o][.]	Negative Absolute	POWER family	XO	31	488
divw[o][.]	Divide Word	PowerPC	XO	31	491
cli	Cache Line Invalidate	POWER family	X	31	502
mcrxr	Move to Condition Register from XER	com	X	31	512
clcs	Cache Line Compute Size	POWER family	X	31	531
lswx	Load String Word Indexed	PowerPC	X	31	533
lsx	Load String Indexed	POWER family	X	31	533
lbrx	Load Byte-Reversed Indexed	POWER family	X	31	534
lwbrx	Load Word Byte-Reversed Indexed	PowerPC	X	31	534
lfsx	Load Floating-Point Single Indexed	com	X	31	535
sr[.]	Shift Right	POWER family	X	31	536
srw[.]	Shift Right Word	PowerPC	X	31	536
rrib[.]	Rotate Right and Insert Bit	POWER family	X	31	537
maskir[.]	Mask Insert from Register	POWER family	X	31	541

Item	Description	Description	Description	Description	Description
tlbsync	Translation Look-aside Buffer Synchronize	PPC opt.	X	31	566
lfsux	Load Floating-Point Single with Update Indexed	com	X	31	567
mfsr	Move from Segment Register	com	X	31	595
lsi	Load String Immediate	POWER family	X	31	597
lswi	Load String Word Immediate	PowerPC	X	31	597
dcs	Data Cache Synchronize	POWER family	X	31	598
sync	Synchronize	PowerPC	X	31	598
lfdx	Load Floating-Point Double Indexed	com	X	31	599
mfsri	Move from Segment Register Indirect	POWER family	X	31	627
dclst	Data Cache Line Store	POWER family	X	31	630
lfdux	Load Floating-Point Double with Update Indexed	com	X	31	631
mfsrin	Move from Segment Register Indirect	PowerPC	X	31	659
stswx	Store String Word Indexed	PowerPC	X	31	661
stsx	Store String Indexed	POWER family	X	31	661
stbrx	Store Byte-Reversed Indexed	POWER family	X	31	662
stwbrx	Store Word Byte-Reversed Indexed	PowerPC	X	31	662
stfsx	Store Floating-Point Single Indexed	com	X	31	663
srq[.]	Shift Right with MQ	POWER family	X	31	664
sre[.]	Shift Right Extended	POWER family	X	31	665
stfsux	Store Floating-Point Single with Update Indexed	com	X	31	695
sriq[.]	Shift Right Immediate with MQ	POWER family	X	31	696
stsi	Store String Immediate	POWER family	X	31	725
stswi	Store String Word Immediate	PowerPC	X	31	725
stfdx	Store Floating-Point Double Indexed	com	X	31	727
srlq[.]	Shift Right Long with MQ	POWER family	X	31	728
sreq[.]	Shift Right Extended with MQ	POWER family	X	31	729

Item	Description	Description	Description	Description	Description
stfdux	Store Floating-Point Double with Update Indexed	com	X	31	759
srliq[.]	Shift Right Long Immediate with MQ	POWER family	X	31	760
lhbrx	Load Half Byte-Reversed Indexed	com	X	31	790
lfqx	Load Floating-Point Quad Indexed	POWER2™	X	31	791
sra[.]	Shift Right Algebraic	POWER family	X	31	792
sraw[.]	Shift Right Algebraic Word	PowerPC	X	31	792
rac[.]	Real Address Compute	POWER family	X	31	818
lfqux	Load Floating-Point Quad with Update Indexed	POWER2™	X	31	823
srai[.]	Shift Right Algebraic Immediate	POWER family	X	31	824
srawi[.]	Shift Right Algebraic Word Immediate	PowerPC	X	31	824
eieio	Enforce In-order Execution of I/O	PowerPC	X	31	854
sthbrx	Store Half Byte-Reverse Indexed	com	X	31	918
stfqx	Store Floating-Point Quad Indexed	POWER2™	X	31	919
sraq[.]	Shift Right Algebraic with MQ	POWER family	X	31	920
srea[.]	Shift Right Extended Algebraic	POWER family	X	31	921
exts[.]	Extend Sign	POWER family	X	31	922
extsh[.]	Extend Sign Halfword	PowerPC	XO	31	922
stfqux	Store Floating-Point Quad with Update Indexed	POWER2™	X	31	951
sraiql[.]	Shift Right Algebraic Immediate with MQ	POWER family	X	31	952
extsb[.]	Extend Sign Byte	PowerPC	X	31	954
tlbld	Load Data TLB Entry	603 only	X	31	978
icbi	Instruction Cache Block Invalidate	PowerPC	X	31	982
stfiwx	Store Floating-Point as Integer Word Indexed	PPC opt.	X	31	983
tlbli	Load Instruction TLB Entry	603 only	X	31	1010

Item	Description	Description	Description	Description	Description
dcbz	Data Cache Block Set to Zero	PowerPC	X	31	1014
dclz	Data Cache Line Set to Zero	POWER family	X	31	1014
l	Load	POWER family	D	32	
lwz	Load Word and Zero	PowerPC	D	32	
lu	Load with Update	POWER family	D	33	
lwzu	Load Word with Zero Update	PowerPC	D	33	
lbz	Load Byte and Zero	com	D	34	
lbzu	Load Byte and Zero with Update	com	D	35	
st	Store	POWER family	D	36	
stw	Store	PowerPC	D	36	
stu	Store with Update	POWER family	D	37	
stwu	Store Word with Update	PowerPC	D	37	
stb	Store Byte	com	D	38	
lhz	Load Half and Zero	com	D	40	
lhzu	Load Half and Zero with Update	com	D	41	
lha	Load Half Algebraic	com	D	42	
lhau	Load Half Algebraic with Update	com	D	43	
sth	Store Half	com	D	44	
sthu	Store Half with Update	com	D	45	
lm	Load Multiple	POWER family	D	46	
lmw	Load Multiple Word	PowerPC	D	46	
stm	Store Multiple	POWER family	D	47	
stmw	Store Multiple Word	PowerPC	D	47	
lfs	Load Floating-Point Single	com	D	48	
lfsu	Load Floating-Point Single with Update	com	D	49	
lfd	Load Floating-Point Double	com	D	50	
lfdu	Load Floating-Point Double with Update	com	D	51	
stfs	Store Floating-Point Single	com	D	52	
stfsu	Store Floating-Point Single with Update	com	D	53	
stfd	Store Floating-Point Double	com	D	54	
stfdu	Store Floating-Point Double with Update	com	D	55	
lfq	Load Floating-Point Quad	POWER2™	D	56	
lfqu	Load Floating-Point Quad with Update	POWER2™	D	57	

Item	Description	Description	Description	Description	Description
fdivs[.]	Floating Divide Single	PowerPC	A	59	18
fsubs[.]	Floating Subtract Single	PowerPC	A	59	20
fadds[.]	Floating Add Single	PowerPC	A	59	21
fres[.]	Floating Reciprocal Estimate Single	PPC opt.	A	59	24
fmuls[.]	Floating Multiply Single	PowerPC	A	59	25
fmsubs[.]	Floating Multiply-Subtract Single	PowerPC	A	59	28
fmadds[.]	Floating Multiply-Add Single	PowerPC	A	59	29
fnmsubs[.]	Floating Negative Multiply-Subtract Single	PowerPC	A	59	30
fnmadds[.]	Floating Negative Multiply-Add Single	PowerPC	A	59	31
stfq	Store Floating-Point Quad	POWER2™	DS	60	
stfqu	Store Floating-Point Quad with Update	POWER2™	DS	61	
fcmpu	Floating Compare Unordered	com	XL	63	0
frsp[.]	Floating Round to Single Precision	com	X	63	12
fcir[.]	Floating Convert to Integer Word	POWER family	X	63	14
fctiw[.]	Floating Convert to Integer Word	PowerPC	X	63	14
fcirz[.]	Floating Convert to Integer Word with Round to Zero	POWER family	X	63	15
fctiwz[.]	Floating Convert to Integer Word with Round to Zero	PowerPC	XL	63	15
fd[.]	Floating Divide	POWER family	A	63	18
fdiv[.]	Floating Divide	PowerPC	A	63	18
fs[.]	Floating Subtract	POWER family	A	63	20
fsub[.]	Floating Subtract	PowerPC	A	63	20
fa[.]	Floating Add	POWER family	A	63	21
fadd[.]	Floating Add	PowerPC	A	63	21
fsqrt[.]	Floating Square Root	POWER2™	A	63	22
fsel[.]	Floating-Point Select	PPC opt.	A	63	23
fm[.]	Floating Multiply	POWER family	A	63	25
fmul[.]	Floating Multiply	PowerPC	A	63	25
frsqrt[.]	Floating Reciprocal Square Root Estimate	PPC opt.	A	63	26

Item	Description	Description	Description	Description	Description
fms[.]	Floating Multiply-Subtract	POWER family	A	63	28
fmsub[.]	Floating Multiply-Subtract	PowerPC	A	63	28
fma[.]	Floating Multiply-Add	POWER family	A	63	29
fmadd[.]	Floating Multiply-Add	PowerPC	A	63	29
fnms[.]	Floating Negative Multiply-Subtract	POWER family	A	63	30
fnmsub[.]	Floating Negative Multiply-Subtract	PowerPC	A	63	30
fnma[.]	Floating Negative Multiply-Add	POWER family	A	63	31
fnmadd[.]	Floating Negative Multiply-Add	PowerPC	A	63	31
fcmpo	Floating Compare Ordered	com	X	63	32
mtfsb1[.]	Move to FPSCR Bit 1	com	X	63	38
fneg[.]	Floating Negate	com	X	63	40
mcrfs	Move to Condition Register from FPSCR	com	X	63	64
mtfsb0[.]	Move to FPSCR Bit 0	com	X	63	70
fmr[.]	Floating Move Register	com	X	63	72
mtfsfi[.]	Move to FPSCR Field Immediate	com	X	63	134
fnabs[.]	Floating Negative Absolute Value	com	X	63	136
fabs[.]	Floating Absolute Value	com	X	63	264
mffs[.]	Move from FPSCR	com	X	63	583
mtfsf[.]	Move to FPSCR Fields	com	XFL	63	711

Appendix D instructions common to POWER family, POWER2™, and PowerPC

Item	Description	Description	Description	Description
Instructions Common to POWER family, POWER2™, and PowerPC	Instructions Common to POWER family, POWER2™, and PowerPC	Instructions Common to POWER family, POWER2™, and PowerPC	Instructions Common to POWER family, POWER2™, and PowerPC	Instructions Common to POWER family, POWER2™, and PowerPC
Mnemonic	Instruction	Format	Primary Op Code	Extended Op Code
and[.]	AND	X	31	28
andc[.]	AND with Complement	X	31	60
b[l][a]	Branch	I	18	
bc[l][a]	Branch Conditional	B	16	
cmp	Compare	X	31	0

Item	Description	Description	Description	Description
cmpi	Compare Immediate	D	11	
cmpl	Compare Logical	X	31	32
cmpli	Compare Logical Immediate	D	10	
crand	Condition Register AND	XL	19	257
crandc	Condition Register AND with Complement	XL	19	129
creqv	Condition Register Equivalent	XL	19	289
crnand	Condition Register NAND	XL	19	225
crnor	Condition Register NOR	XL	19	33
cror	Condition Register OR	XL	19	449
crorc	Condition Register OR with Complement	XL	19	417
crxor	Condition Register XOR	XL	19	193
eciwx	External Control in Word Indexed	X	31	310
ecowx	External Control out Word Indexed	X	31	438
eqv[.]	Equivalent	X	31	284
fabs[.]	Floating Absolute Value	X	63	264
fcmpo	Floating Compare Ordered	X	63	32
fcmpu	Floating Compare Unordered	XL	63	0
fmr[.]	Floating Move Register	X	63	72
fnabs[.]	Floating Negative Absolute Value	X	63	136
fneg[.]	Floating Negate	X	63	40
frsp[.]	Floating Round to Single Precision	X	63	12
lbz	Load Byte and Zero	D	34	
lbzu	Load Byte and Zero with Update	D	35	
lbzux	Load Byte and Zero with Update Indexed	X	31	119
lbzx	Load Byte and Zero Indexed	X	31	87
lfd	Load Floating-Point Double	D	50	
lfdu	Load Floating-Point Double with Update	D	51	
lfdux	Load Floating-Point Double with Update Indexed	X	31	631
lfdx	Load Floating-Point Double Indexed	X	31	599
lfs	Load Floating-Point Single	D	48	
lfsu	Load Floating-Point Single with Update	D	49	

Item	Description	Description	Description	Description
lfsux	Load Floating-Point Single with Update Indexed	X	31	567
lfsx	Load Floating-Point Single Indexed	X	31	535
lha	Load Half Algebraic	D	42	
lhau	Load Half Algebraic with Update	D	43	
lhaux	Load Half Algebraic with Update Indexed	X	31	375
lhax	Load Half Algebraic Indexed	X	31	343
lhbrx	Load Half Byte-Reversed Indexed	X	31	790
lhz	Load Half and Zero	D	40	
lhzu	Load Half and Zero with Update	D	41	
lhzux	Load Half and Zero with Update Indexed	X	31	331
lhzx	Load Half and Zero Indexed	X	31	279
mcrf	Move Condition Register Field	XL	19	0
mcrfs	Move to Condition Register from FPSCR	X	63	64
mcrxr	Move to Condition Register from XER	X	31	512
mfcrr	Move from Condition Register	X	31	19
mffs[.]	Move from FPSCR	X	63	583
mfmsr	Move from Machine State Register	X	31	83
mf spr	Move from Special-Purpose Register	X	31	339
mfsr	Move from Segment Register	X	31	595
mtcrf	Move to Condition Register Fields	XFX	31	144
mtfsb0[.]	Move to FPSCR Bit 0	X	63	70
mtfsb1[.]	Move to FPSCR Bit 1	X	63	38
mtfsf[.]	Move to FPSCR Fields	XFL	63	711
mtfsfi[.]	Move to FPSCR Field Immediate	X	63	134
mtmsr	Move to Machine State Register	X	31	146
mt spr	Move to Special-Purpose Register	X	31	467
mtsr	Move to Segment Register	X	31	210
nand[.]	NAND	X	31	476
neg[o][.]	Negate	XO	31	104
nor[.]	NOR	X	31	124
or[.]	OR	X	31	444

Item	Description	Description	Description	Description
orc[.]	OR with Complement	X	31	412
rfi	Return from Interrupt	X	19	50
si	Subtract Immediate	D	12	
si.	Subtract Immediate and Record	D	13	
stb	Store Byte	D	38	
stbu	Store Byte with Update	D	39	
stbux	Store Byte with Update Indexed	X	31	247
stbx	Store Byte Indexed	X	31	215
stfd	Store Floating-Point Double	D	54	
stfdu	Store Floating-Point Double with Update	D	55	
stfdux	Store Floating-Point Double with Update Indexed	X	31	759
stfdx	Store Floating-Point Double Indexed	X	31	727
stfs	Store Floating-Point Single	D	52	
stfsu	Store Floating-Point Single with Update	D	53	
stfsux	Store Floating-Point Single with Update Indexed	X	31	695
stfsx	Store Floating-Point Single Indexed	X	31	663
sth	Store Half	D	44	
sthbrx	Store Half Byte-Reverse Indexed	X	31	918
sthu	Store Half with Update	D	45	
sthux	Store Half with Update Indexed	X	31	439
sthx	Store Half Indexed	X	31	407
xor[.]	XOR	X	31	316

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:

Item	Description	Description	Description	Description	Description
POWER family and POWER2™ Instructions	POWER family and POWER2™ Instructions	POWER family and POWER2™ Instructions	POWER family and POWER2™ Instructions	POWER family and POWER2™ Instructions	POWER family and POWER2™ Instructions
Mnemonic	Instruction	POWER2™ Only	Format	Primary Op Code	Extended Op Code
a[o][.]	Add Carrying		XO	31	10
abs[o][.]	Absolute		XO	31	360
ae[o][.]	Add Extended		XO	31	138
ai	Add Immediate		D	12	
ai.	Add Immediate and Record		D	13	
ame[o][.]	Add to Minus One Extended		XO	31	234
and[.]	AND		X	31	28
andc[.]	AND with Complement		X	31	60
andil.	AND Immediate Lower		D	28	
andiu.	AND Immediate Upper		D	29	
aze[o][.]	Add to Zero Extended		XO	31	202
b[l][a]	Branch		I	18	
bc[l][a]	Branch Conditional		B	16	
bcc[l]	Branch Conditional to Count Register		XL	19	528
bcr[l]	Branch Conditional Register		XL	19	16
cal	Compute Address Lower		D	14	
cau	Compute Address Upper		D	15	
cax[o][.]	Compute Address		XO	31	266
clcs	Cache Line Compute Size		X	31	531
clf	Cache Line Flush		X	31	118
cli	Cache Line Invalidate		X	31	502
cmp	Compare		X	31	0
cmpi	Compare Immediate		D	11	
cmpl	Compare Logical		X	31	32
cmpli	Compare Logical Immediate		D	10	
cntlz[.]	Count Leading Zeros		X	31	26
crand	Condition Register AND		XL	19	257
crandc	Condition Register AND with Complement		XL	19	129
creqv	Condition Register Equivalent		XL	19	289

Item	Description	Description	Description	Description	Description
crnand	Condition Register NAND		XL	19	225
crnor	Condition Register NOR		XL	19	33
cror	Condition Register OR		XL	19	449
crorc	Condition Register OR with Complement		XL	19	417
crxor	Condition Register XOR		XL	19	193
dclst	Data Cache Line Store		X	31	630
dclz	Data Cache Line Set to Zero		X	31	1014
dcs	Data Cache Synchronize		X	31	598
div[o][.]	Divide		XO	31	331
divs[o][.]	Divide Short		XO	31	363
doz[o][.]	Difference or Zero		XO	31	264
dozi	Difference or Zero Immediate		D	09	
eciwx	External Control in Word Indexed		X	31	310
ecowx	External Control out Word Indexed		X	31	438
eqv[.]	Equivalent		X	31	284
exts[.]	Extend Sign		X	31	922
fa[.]	Floating Add		A	63	21
fabs[.]	Floating Absolute Value		X	63	264
fcir[.]	Floating Convert to Integer Word		X	63	14
fcirz[.]	Floating Convert to Integer Word with Round to Zero		X	63	15
fcmpo	Floating Compare Ordered		X	63	32
fcmpu	Floating Compare Unordered		XL	63	0
fd[.]	Floating Divide		A	63	18
fm[.]	Floating Multiply		A	63	25
fma[.]	Floating Multiply-Add		A	63	29
fmr[.]	Floating Move Register		X	63	72
fms[.]	Floating Multiply-Subtract		A	63	28
fnabs[.]	Floating Negative Absolute Value		X	63	136
fneg[.]	Floating Negate		X	63	40
fnma[.]	Floating Negative Multiply-Add		A	63	31

Item	Description	Description	Description	Description	Description
fnms[.]	Floating Negative Multiply-Subtract		A	63	30
frsp[.]	Floating Round to Single Precision		X	63	12
fs[.]	Floating Subtract		A	63	20
fsqrt[.]	Floating Square Root	POWER2™	A	63	22
ics	Instruction Cache Synchronize		X	19	150
l	Load		D	32	
lbrx	Load Byte-Reversed Indexed		X	31	534
lbz	Load Byte and Zero		D	34	
lbzu	Load Byte and Zero with Update		D	35	
lbzux	Load Byte and Zero with Update Indexed		X	31	119
lbzx	Load Byte and Zero Indexed		X	31	87
lfd	Load Floating-Point Double		D	50	
lfd�	Load Floating-Point Double with Update		D	51	
lfdûx	Load Floating-Point Double with Update Indexed		X	31	631
lfdx	Load Floating-Point Double Indexed		X	31	599
lfq	Load Floating-Point Quad	POWER2™	D	56	
lfqu	Load Floating-Point Quad with Update	POWER2™	D	57	
lfqux	Load Floating-Point Quad with Update Indexed	POWER2™	X	31	823
lfqx	Load Floating-Point Quad Indexed	POWER2™	X	31	791
lfs	Load Floating-Point Single		D	48	
lfsu	Load Floating-Point Single with Update		D	49	
lfsux	Load Floating-Point Single with Update Indexed		X	31	567
lfsx	Load Floating-Point Single Indexed		X	31	535
lha	Load Half Algebraic		D	42	
lhau	Load Half Algebraic with Update		D	43	
lhaux	Load Half Algebraic with Update Indexed		X	31	375

Item	Description	Description	Description	Description	Description
lhax	Load Half Algebraic Indexed		X	31	343
lhbrx	Load Half Byte-Reversed Indexed		X	31	790
lhz	Load Half and Zero		D	40	
lhzu	Load Half and Zero with Update		D	41	
lhzux	Load Half and Zero with Update Indexed		X	31	331
lhzx	Load Half and Zero Indexed		X	31	279
lm	Load Multiple		D	46	
lscbx	Load String and Compare Byte Indexed		X	31	277
lsi	Load String Immediate		X	31	597
lsx	Load String Indexed		X	31	533
lu	Load with Update		D	33	
lux	Load with Update Indexed		X	31	55
lx	Load Indexed		X	31	23
maskg[.]	Mask Generate		X	31	29
maskir[.]	Mask Insert from Register		X	31	541
mcrf	Move Condition Register Field		XL	19	0
mcrfs	Move to Condition Register from FPSCR		X	63	64
mcrxr	Move to Condition Register from XER		X	31	512
mfcrr	Move from Condition Register		X	31	19
mffs[.]	Move from FPSCR		X	63	583
mfmsr	Move from Machine State Register		X	31	83
mfsprr	Move from Special-Purpose Register		X	31	339
mfsrr	Move from Segment Register		X	31	595
mfsrri	Move from Segment Register Indirect		X	31	627
mtcrf	Move to Condition Register Fields		XFX	31	144
mtfsb0[.]	Move to FPSCR Bit 0		X	63	70
mtfsb1[.]	Move to FPSCR Bit 1		X	63	38
mtfsf[.]	Move to FPSCR Fields		XFL	63	711

Item	Description	Description	Description	Description	Description
mtfsfi[.]	Move to FPSCR Field Immediate		X	63	134
mtmsr	Move to Machine State Register		X	31	146
mtspr	Move to Special-Purpose Register		X	31	467
mtsr	Move to Segment Register		X	31	210
mtsri	Move to Segment Register Indirect		X	31	242
mul[o][.]	Multiply		XO	31	107
muli	Multiply Immediate		D	07	
muls[o][.]	Multiply Short		XO	31	235
nabs[o][.]	Negative Absolute		XO	31	488
nand[.]	NAND		X	31	476
neg[o][.]	Negate		XO	31	104
nor[.]	NOR		X	31	124
or[.]	OR		X	31	444
orc[.]	OR with Complement		X	31	412
oril	OR Immediate Lower		D	24	
oriu	OR Immediate Upper		D	25	
rac[.]	Real Address Compute		X	31	818
rfi	Return from Interrupt		X	19	50
rfsvc	Return from SVC		X	19	82
rlimi[.]	Rotate Left Immediate then Mask Insert		M	20	
rlinm[.]	Rotate Left Immediate then AND with Mask		M	21	
rlmi[.]	Rotate Left then Mask Insert		M	22	
rlnm[.]	Rotate Left then AND with Mask		M	23	
rrib[.]	Rotate Right and Insert Bit		X	31	537
sf[o][.]	Subtract from		XO	31	08
sfe[o][.]	Subtract from Extended		XO	31	136
sfi	Subtract from Immediate		D	08	
sfme[o][.]	Subtract from Minus One Extended		XO	31	232
sfze[o][.]	Subtract from Zero Extended		XO	31	200
si	Subtract Immediate		D	12	

Item	Description	Description	Description	Description	Description
si.	Subtract Immediate and Record		D	13	
sl[.]	Shift Left		X	31	24
sle[.]	Shift Left Extended		X	31	153
sleq[.]	Shift Left Extended with MQ		X	31	217
sliq[.]	Shift Left Immediate with MQ		X	31	184
slliq[.]	Shift Left Long Immediate with MQ		X	31	248
sllq[.]	Shift Left Long with MQ		X	31	216
slq[.]	Shift Left with MQ		X	31	152
sr[.]	Shift Right		X	31	536
sra[.]	Shift Right Algebraic		X	31	792
srai[.]	Shift Right Algebraic Immediate		X	31	824
sraiq[.]	Shift Right Algebraic Immediate with MQ		X	31	952
sraq[.]	Shift Right Algebraic with MQ		X	31	920
sre[.]	Shift Right Extended		X	31	665
srea[.]	Shift Right Extended Algebraic		X	31	921
sreq[.]	Shift Right Extended with MQ		X	31	729
sriq[.]	Shift Right Immediate with MQ		X	31	696
srlmq[.]	Shift Right Long Immediate with MQ		X	31	760
srlq[.]	Shift Right Long with MQ		X	31	728
srq[.]	Shift Right with MQ		X	31	664
st	Store		D	36	
stb	Store Byte		D	38	
stbrx	Store Byte-Reversed Indexed		X	31	662
stbu	Store Byte with Update		D	39	
stbux	Store Byte with Update Indexed		X	31	247
stbx	Store Byte Indexed		X	31	215
stfd	Store Floating-Point Double		D	54	
stfdu	Store Floating-Point Double with Update		D	55	
stfdux	Store Floating-Point Double with Update Indexed		X	31	759

Item	Description	Description	Description	Description	Description
stfdx	Store Floating-Point Double Indexed		X	31	727
stfq	Store Floating-Point Quad	POWER2™	DS	60	
stfqu	Store Floating-Point Quad with Update	POWER2™	DS	61	
stfqux	Store Floating-Point Quad with Update Indexed	POWER2™	X	31	951
stfqx	Store Floating-Point Quad Indexed	POWER2™	X	31	919
stfs	Store Floating-Point Single		D	52	
stfsu	Store Floating-Point Single with Update		D	53	
stfsux	Store Floating-Point Single with Update Indexed		X	31	695
stfsx	Store Floating-Point Single Indexed		X	31	663
sth	Store Half		D	44	
sthbrx	Store Half Byte-Reverse Indexed		X	31	918
sthu	Store Half with Update		D	45	
sthux	Store Half with Update Indexed		X	31	439
sthx	Store Half Indexed		X	31	407
stm	Store Multiple		D	47	
stsi	Store String Immediate		X	31	725
stsx	Store String Indexed		X	31	661
stu	Store with Update		D	37	
stux	Store with Update Indexed		X	31	183
stx	Store Indexed		X	31	151
svc[l][a]	Supervisor Call		SC	17	
t	Trap		X	31	04
ti	Trap Immediate		D	03	
tlbi	Translation Look-aside Buffer Invalidate Entry		X	31	306
xor[.]	XOR		X	31	316
xoril	XOR Immediate Lower		D	26	
xoriu	XOR Immediate Upper		D	27	

Appendix F PowerPC® instructions

Table 40. PowerPC® Instructions

Mnemonic	Instruction	Format	Primary Op Code	Extended Op Code
add[o][.]	Add	XO	31	266
addc[o][.]	Add Carrying	XO	31	10
adde[o][.]	Add Extended	XO	31	138
addi	Add Immediate	D	14	
addic	Add Immediate Carrying	D	12	
addic.	Add Immediate Carrying and Record	D	13	
addis	Add Immediate Shifted	D	15	
addme[o][.]	Add to Minus One Extended	XO	31	234
addze[o][.]	Add to Zero Extended	XO	31	202
and[.]	AND	X	31	28
andc[.]	AND with Complement	X	31	60
andi.	AND Immediate	D	28	
andis.	AND Immediate Shifted	D	29	
b[l][a]	Branch	I	18	
bc[l][a]	Branch Conditional	B	16	
bcctr[l]	Branch Conditional to Count Register	XL	19	528
bclr[l]	Branch Conditional Link Register	XL	19	16
cmp	Compare	X	31	0
cmpi	Compare Immediate	D	11	
cmpl	Compare Logical	X	31	32
cmpli	Compare Logical Immediate	D	10	
cntlzd	Count Leading Zeros Doubleword	X	31	58
cntlzw[.]	Count Leading Zeros Word	X	31	26
crand	Condition Register AND	XL	19	257
crandc	Condition Register AND with Complement	XL	19	129
creqv	Condition Register Equivalent	XL	19	289
crnand	Condition Register NAND	XL	19	225
crnor	Condition Register NOR	XL	19	33
cror	Condition Register OR	XL	19	449
crorc	Condition Register OR with Complement	XL	19	417
crxor	Condition Register XOR	XL	19	193
dcbf	Data Cache Block Flush	X	31	86
dcbi	Data Cache Block Invalidate	X	31	470
dcbst	Data Cache Block Store	X	31	54
dcbt	Data Cache Block Touch	X	31	278

Table 40. PowerPC® Instructions (continued)

Mnemonic	Instruction	Format	Primary Op Code	Extended Op Code
dcbtst	Data Cache Block Touch for Store	X	31	246
dcbz	Data Cache Block Set to Zero	X	31	1014
divd	Divide Doubleword	XO	31	489
divdu	Divide Doubleword Unsigned	XO	31	457
divw[o].]	Divide Word	XO	31	491
divwu[o].]	Divide Word Unsigned	XO	31	459
eciwx	External Control in Word Indexed (opt.)	X	31	310
ecowx	External Control out Word Indexed (opt.)	X	31	438
eieio	Enforce In-order Execution of I/O	X	31	854
eqv[.]	Equivalent	X	31	284
extsb[.]	Extend Sign Byte	X	31	954
extsh[.]	Extend Sign Halfword	XO	31	922
extsw	Extend Sign Word	X	31	986
fabs[.]	Floating Absolute Value	X	63	264
fadd[.]	Floating Add	A	63	21
fadds[.]	Floating Add Single	A	59	21
fcfid	Floating Convert from Integer Doubleword	X	63	846
fcmpo	Floating Compare Ordered	X	63	32
fcmpu	Floating Compare Unordered	XL	63	0
fctid	Floating Convert to Integer Doubleword	X	63	814
fctidz	Floating Convert to Integer Doubleword with Round Toward Zero	X	63	815
fctiw[.]	Floating Convert to Integer Word	X	63	14
fctiwz[.]	Floating Convert to Integer Word with Round to Zero	XL	63	15
fdiv[.]	Floating Divide	A	63	18
fdivs[.]	Floating Divide Single	A	59	18
fmadd[.]	Floating Multiply-Add	A	63	29
fmadds[.]	Floating Multiply-Add Single	A	59	29
fmr[.]	Floating Move Register	X	63	72
fmsub[.]	Floating Multiply-Subtract	A	63	28
fmsubs[.]	Floating Multiply-Subtract Single	A	59	28
fmul[.]	Floating Multiply	A	63	25
fmuls[.]	Floating Multiply Single	A	59	25

Table 40. PowerPC® Instructions (continued)

Mnemonic	Instruction	Format	Primary Op Code	Extended Op Code
fnabs[.]	Floating Negative Absolute Value	X	63	136
fneg[.]	Floating Negate	X	63	40
fnmadd[.]	Floating Negative Multiply-Add	A	63	31
fnmadds[.]	Floating Negative Multiply-Add Single	A	59	31
fnmsub[.]	Floating Negative Multiply-Subtract	A	63	30
fnmsubs[.]	Floating Negative Multiply-Subtract Single	A	59	30
fres[.]	Floating Reciprocal Estimate Single (optional)	A	59	24
frsp[.]	Floating Round to Single Precision	X	63	12
frsqste[.]	Floating Reciprocal Square Root Estimate (optional)	A	63	26
fsel[.]	Floating-Point Select (optional)	A	63	23
fsub[.]	Floating Subtract	A	63	20
fsubs[.]	Floating Subtract Single	A	59	20
icbi	Instruction Cache Block Invalidate	X	31	982
isync	Instruction Synchronize	X	19	150
lbz	Load Byte and Zero	D	34	
lbzu	Load Byte and Zero with Update	D	35	
lbzux	Load Byte and Zero with Update Indexed	X	31	119
lbzx	Load Byte and Zero Indexed	X	31	87
ld	Load Doubleword	DS	58	0
ldarx	Load Doubleword and Reserve Indexed	X	31	84
ldu	Load Doubleword with Update	DS	58	1
ldux	Load Doubleword with Update Indexed	X	31	53
ldx	Load Doubleword Indexed	X	31	21
lfd	Load Floating-Point Double	D	50	
lfdu	Load Floating-Point Double with Update	D	51	
lfdux	Load Floating-Point Double with Update Indexed	X	31	631
lfdx	Load Floating-Point Double Indexed	X	31	599
lfs	Load Floating-Point Single	D	48	

Table 40. PowerPC® Instructions (continued)

Mnemonic	Instruction	Format	Primary Op Code	Extended Op Code
lfsu	Load Floating-Point Single with Update	D	49	
lfsux	Load Floating-Point Single with Update Indexed	X	31	567
lfsx	Load Floating-Point Single Indexed	X	31	535
lha	Load Half Algebraic	D	42	
lhau	Load Half Algebraic with Update	D	43	
lhaux	Load Half Algebraic with Update Indexed	X	31	375
lhax	Load Half Algebraic Indexed	X	31	343
lhbrx	Load Half Byte-Reversed Indexed	X	31	790
lhz	Load Half and Zero	D	40	
lhzu	Load Half and Zero with Update	D	41	
lhzux	Load Half and Zero with Update Indexed	X	31	331
lhzx	Load Half and Zero Indexed	X	31	279
lmw	Load Multiple Word	D	46	
lswi	Load String Word Immediate	X	31	597
lswx	Load String Word Indexed	X	31	533
lwa	Load Word Algebraic	DS	58	2
lwarx	Load Word and Reserve Indexed	X	31	20
lwaux	Load Word Algebraic with Update Indexed	X	31	373
lwax	Load Word Algebraic Indexed	X	31	341
lwbrx	Load Word Byte-Reversed Indexed	X	31	534
lwz	Load Word and Zero	D	32	
lwzu	Load Word with Zero Update	D	33	
lwzux	Load Word and Zero with Update Indexed	X	31	55
lwzx	Load Word and Zero Indexed	X	31	23
mcrf	Move Condition Register Field	XL	19	0
mcrfs	Move to Condition Register from FPSCR	X	63	64
mcrxr	Move to Condition Register from XER	X	31	512
mfcrr	Move from Condition Register	X	31	19

Table 40. PowerPC® Instructions (continued)

Mnemonic	Instruction	Format	Primary Op Code	Extended Op Code
mffs[.]	Move from FPSCR	X	63	583
mfmsr	Move from Machine State Register	X	31	83
mfspr	Move from Special-Purpose Register	X	31	339
mfsr	Move from Segment Register	X	31	595
mfsrin	Move from Segment Register Indirect	X	31	659
mtcrf	Move to Condition Register Fields	XFX	31	144
mtfsb0[.]	Move to FPSCR Bit 0	X	63	70
mtfsb1[.]	Move to FPSCR Bit 1	X	63	38
mtfsf[.]	Move to FPSCR Fields	XFL	63	711
mtfsfi[.]	Move to FPSCR Field Immediate	X	63	134
mtmsr	Move to Machine State Register	X	31	146
mtspr	Move to Special-Purpose Register	X	31	467
mtsr	Move to Segment Register	X	31	210
mtsrin	Move to Segment Register Indirect	X	31	242
mulhd	Multiply High Doubleword	XO	31	73
mulhdu	Multiply High Doubleword Unsigned	XO	31	9
mulhw[.]	Multiply High Word	XO	31	75
mulhwu[.]	Multiply High Word Unsigned	XO	31	11
mulld	Multiply Low Doubleword	XO	31	233
mulldi	Multiply Low Immediate	D	07	
mulldw[o][.]	Multiply Low Word	XO	31	235
nand[.]	NAND	X	31	476
neg[o][.]	Negate	XO	31	104
nor[.]	NOR	X	31	124
or[.]	OR	X	31	444
orc[.]	OR with Complement	X	31	412
ori	OR Immediate	D	24	
oris	OR Immediate Shifted	D	25	
rfi	Return from Interrupt	X	19	50
rldcl	Rotate Left Doubleword then Clear Left	MDS	30	8
rldcr	Rotate Left Doubleword then Clear Right	MDS	30	9
rldic	Rotate Left Doubleword Immediate then Clear	MD	30	2

Table 40. PowerPC® Instructions (continued)

Mnemonic	Instruction	Format	Primary Op Code	Extended Op Code
rldicl	Rotate Left Doubleword Immediate then Clear Left	MD	30	0
rldicr	Rotate Left Doubleword Immediate then Clear Right	MD	30	1
rldimi	Rotate Left Doubleword Immediate then Mask Insert	MD	30	3
rlwimi[.]	Rotate Left Word Immediate then Mask Insert	M	20	
rlwinm[.]	Rotate Left Word Immediate then AND with Mask	M	21	
rlwnm[.]	Rotate Left Word then AND with Mask	M	23	
sc	System Call	SC	17	
si	Subtract Immediate	D	12	
si.	Subtract Immediate and Record	D	13	
slbia	SLB Invalidate All	X	31	498
slbie	SLB Invalidate Entry	X	31	434
sld	Shift Left Doubleword	X	31	27
slw[.]	Shift Left Word	X	31	24
srad	Shift Right Algebraic Doubleword	X	31	794
sradi	Shift Right Algebraic Doubleword Immediate	XS	31	413
srd	Shift Right Doubleword	X	31	539
sraw[.]	Shift Right Algebraic Word	X	31	792
srawi[.]	Shift Right Algebraic Word Immediate	X	31	824
srw[.]	Shift Right Word	X	31	536
stb	Store Byte	D	38	
stbu	Store Byte with Update	D	39	
stbux	Store Byte with Update Indexed	X	31	247
stbx	Store Byte Indexed	X	31	215
std	Store Doubleword	DS	62	0
stdcx	Store Doubleword Conditional Indexed	X	31	214
stdu	Store Doubleword with Update	DS	62	1
stdux	Store Doubleword with Update Indexed	X	31	181
stdx	Store Doubleword Indexed	X	31	149
stfd	Store Floating-Point Double	D	54	

Table 40. PowerPC® Instructions (continued)

Mnemonic	Instruction	Format	Primary Op Code	Extended Op Code
stfdu	Store Floating-Point Double with Update	D	55	
stfdx	Store Floating-Point Double with Update Indexed	X	31	759
stfdx	Store Floating-Point Double Indexed	X	31	727
stfiwx	Store Floating-Point as Integer Word Indexed (optional)	X	31	983
stfs	Store Floating-Point Single	D	52	
stfsu	Store Floating-Point Single with Update	D	53	
stfsux	Store Floating-Point Single with Update Indexed	X	31	695
stfsx	Store Floating-Point Single Indexed	X	31	663
sth	Store Half	D	44	
sthbrx	Store Half Byte-Reverse Indexed	X	31	918
sthu	Store Half with Update	D	45	
sthux	Store Half with Update Indexed	X	31	439
sthx	Store Half Indexed	X	31	407
stmw	Store Multiple Word	D	47	
stswi	Store String Word Immediate	X	31	725
stswx	Store String Word Indexed	X	31	661
stw	Store	D	36	
stwbrx	Store Word Byte-Reversed Indexed	X	31	662
stwcx.	Store Word Conditional Indexed	X	31	150
stwu	Store Word with Update	D	37	
stwux	Store Word with Update Indexed	X	31	183
stwx	Store Word Indexed	X	31	151
subf[o][.]	Subtract from	XO	31	40
subfc[o][.]	Subtract from Carrying	XO	31	08
subfe[o][.]	Subtract from Extended	XO	31	136
subfic	Subtract from Immediate Carrying	D	08	
subfme[o][.]	Subtract from Minus One Extended	XO	31	232
subfze[o][.]	Subtract from Zero Extended	XO	31	200
sync	Synchronize	X	31	598
td	Trap Doubleword	X	31	68

Table 40. PowerPC® Instructions (continued)

Mnemonic	Instruction	Format	Primary Op Code	Extended Op Code
tdi	Trap Doubleword Immediate	D	2	
tlbie	Translation Look-aside Buffer Invalidate Entry (optional)	X	31	306
tlbsync	Translation Look-aside Buffer Synchronize (optional)	X	31	566
tw	Trap Word	X	31	04
twi	Trap Word Immediate	D	03	
xor[.]	XOR	X	31	316
xori	XOR Immediate	D	26	
xoris	XOR Immediate Shift	D	27	

Appendix G PowerPC 601 RISC Microprocessor instructions

Item	Description	Description	Description	Description
PowerPC 601 RISC Microprocessor Instructions	PowerPC 601 RISC Microprocessor Instructions	PowerPC 601 RISC Microprocessor Instructions	PowerPC 601 RISC Microprocessor Instructions	PowerPC 601 RISC Microprocessor Instructions
Mnemonic	Instruction	Format	Primary Op Code	Extended Op Code
a[o][.]	Add Carrying	XO	31	10
abs[o][.]	Absolute	XO	31	360
add[o][.]	Add	XO	31	266
addc[o][.]	Add Carrying	XO	31	10
adde[o][.]	Add Extended	XO	31	138
addi	Add Immediate	D	14	
addic	Add Immediate Carrying	D	12	
addic.	Add Immediate Carrying and Record	D	13	
addis	Add Immediate Shifted	D	15	
addme[o][.]	Add to Minus One Extended	XO	31	234
addze[o][.]	Add to Zero Extended	XO	31	202
ae[o][.]	Add Extended	XO	31	138
ai	Add Immediate	D	12	
ai.	Add Immediate and Record	D	13	
ame[o][.]	Add to Minus One Extended	XO	31	234
and[.]	AND	X	31	28
andc[.]	AND with Complement	X	31	60
andi.	AND Immediate	D	28	
andil.	AND Immediate Lower	D	28	
andis.	AND Immediate Shifted	D	29	
andiu.	AND Immediate Upper	D	29	
aze[o][.]	Add to Zero Extended	XO	31	202

Item	Description	Description	Description	Description
b[l][a]	Branch	I	18	
bc[l][a]	Branch Conditional	B	16	
bcc[l]	Branch Conditional to Count Register	XL	19	528
bcctr[l]	Branch Conditional to Count Register	XL	19	528
bclr[l]	Branch Conditional Link Register	XL	19	16
bcr[l]	Branch Conditional Register	XL	19	16
cal	Compute Address Lower	D	14	
cau	Compute Address Upper	D	15	
cax[o][.]	Compute Address	XO	31	266
clcs	Cache Line Compute Size	X	31	531
cmp	Compare	X	31	0
cmpi	Compare Immediate	D	11	
cmpl	Compare Logical	X	31	32
cmpli	Compare Logical Immediate	D	10	
cntlz[.]	Count Leading Zeros	X	31	26
cntlzw[.]	Count Leading Zeros Word	X	31	26
crand	Condition Register AND	XL	19	257
crandc	Condition Register AND with Complement	XL	19	129
creqv	Condition Register Equivalent	XL	19	289
crnand	Condition Register NAND	XL	19	225
crnor	Condition Register NOR	XL	19	33
cror	Condition Register OR	XL	19	449
crorc	Condition Register OR with Complement	XL	19	417
crxor	Condition Register XOR	XL	19	193
dcbf	Data Cache Block Flush	X	31	86
dcbi	Data Cache Block Invalidate	X	31	470
dcbst	Data Cache Block Store	X	31	54
dcbt	Data Cache Block Touch	X	31	278
dcbtst	Data Cache Block Touch for Store	X	31	246
dcbz	Data Cache Block Set to Zero	X	31	1014
dcs	Data Cache Synchronize	X	31	598
div[o][.]	Divide	XO	31	331
divs[o][.]	Divide Short	XO	31	363
divw[o][.]	Divide Word	XO	31	491
divwu[o][.]	Divide Word Unsigned	XO	31	459

Item	Description	Description	Description	Description
doz[o][.]	Difference or Zero	XO	31	264
dozi	Difference or Zero Immediate	D	09	
eciwx	External Control in Word Indexed	X	31	310
ecowx	External Control out Word Indexed	X	31	438
eieio	Enforce In-order Execution of I/O	X	31	854
eqv[.]	Equivalent	X	31	284
exts[.]	Extend Sign	X	31	922
extsb[.]	Extend Sign Byte	X	31	954
extsh[.]	Extend Sign Halfword	XO	31	922
fa[.]	Floating Add	A	63	21
fabs[.]	Floating Absolute Value	X	63	264
fadd[.]	Floating Add	A	63	21
fadds[.]	Floating Add Single	A	59	21
fcir[.]	Floating Convert to Integer Word	X	63	14
fcirz[.]	Floating Convert to Integer Word with Round to Zero	X	63	15
fcmpo	Floating Compare Ordered	X	63	32
fcmpu	Floating Compare Unordered	XL	63	0
fctiw[.]	Floating Convert to Integer Word	X	63	14
fctiwz[.]	Floating Convert to Integer Word with Round to Zero	XL	63	15
fd[.]	Floating Divide	A	63	18
fdiv[.]	Floating Divide	A	63	18
fdivs[.]	Floating Divide Single	A	59	18
fm[.]	Floating Multiply	A	63	25
fma[.]	Floating Multiply-Add	A	63	29
fmadd[.]	Floating Multiply-Add	A	63	29
fmadds[.]	Floating Multiply-Add Single	A	59	29
fmr[.]	Floating Move Register	X	63	72
fms[.]	Floating Multiply-Subtract	A	63	28
fmsub[.]	Floating Multiply-Subtract	A	63	28
fmsubs[.]	Floating Multiply-Subtract Single	A	59	28
fmul[.]	Floating Multiply	A	63	25
fmuls[.]	Floating Multiply Single	A	59	25
fnabs[.]	Floating Negative Absolute Value	X	63	136
fneg[.]	Floating Negate	X	63	40

Item	Description	Description	Description	Description
fnma[.]	Floating Negative Multiply-Add	A	63	31
fnmadd[.]	Floating Negative Multiply-Add	A	63	31
fnmadds[.]	Floating Negative Multiply-Add Single	A	59	31
fnms[.]	Floating Negative Multiply-Subtract	A	63	30
fnmsub[.]	Floating Negative Multiply-Subtract	A	63	30
fnmsubs[.]	Floating Negative Multiply-Subtract Single	A	59	30
frsp[.]	Floating Round to Single Precision	X	63	12
fs[.]	Floating Subtract	A	63	20
fsub[.]	Floating Subtract	A	63	20
fsubs[.]	Floating Subtract Single	A	59	20
icbi	Instruction Cache Block Invalidate	X	31	982
ics	Instruction Cache Synchronize	X	19	150
isync	Instruction Synchronize	X	19	150
l	Load	D	32	
lbrx	Load Byte-Reversed Indexed	X	31	534
lbz	Load Byte and Zero	D	34	
lbzu	Load Byte and Zero with Update	D	35	
lbzux	Load Byte and Zero with Update Indexed	X	31	119
lbzx	Load Byte and Zero Indexed	X	31	87
lfd	Load Floating-Point Double	D	50	
lfdu	Load Floating-Point Double with Update	D	51	
lfdux	Load Floating-Point Double with Update Indexed	X	31	631
lfdx	Load Floating-Point Double Indexed	X	31	599
lfs	Load Floating-Point Single	D	48	
lfsu	Load Floating-Point Single with Update	D	49	
lfsux	Load Floating-Point Single with Update Indexed	X	31	567
lfsx	Load Floating-Point Single Indexed	X	31	535
lha	Load Half Algebraic	D	42	
lhau	Load Half Algebraic with Update	D	43	

Item	Description	Description	Description	Description
lhaux	Load Half Algebraic with Update Indexed	X	31	375
lhax	Load Half Algebraic Indexed	X	31	343
lhbrx	Load Half Byte-Reversed Indexed	X	31	790
lhz	Load Half and Zero	D	40	
lhzu	Load Half and Zero with Update	D	41	
lhzux	Load Half and Zero with Update Indexed	X	31	331
lhzx	Load Half and Zero Indexed	X	31	279
lm	Load Multiple	D	46	
lmw	Load Multiple Word	D	46	
lscbx	Load String and Compare Byte Indexed	X	31	277
lsi	Load String Immediate	X	31	597
lswi	Load String Word Immediate	X	31	597
lswx	Load String Word Indexed	X	31	533
lsx	Load String Indexed	X	31	533
lu	Load with Update	D	33	
lux	Load with Update Indexed	X	31	55
lwarx	Load Word and Reserve Indexed	X	31	20
lwbrx	Load Word Byte-Reversed Indexed	X	31	534
lwz	Load Word and Zero	D	32	
lwzu	Load Word with Zero Update	D	33	
lwzux	Load Word and Zero with Update Indexed	X	31	55
lwzx	Load Word and Zero Indexed	X	31	23
lx	Load Indexed	X	31	23
maskg[.]	Mask Generate	X	31	29
maskir[.]	Mask Insert from Register	X	31	541
mcrf	Move Condition Register Field	XL	19	0
mcrfs	Move to Condition Register from FPSCR	X	63	64
mcrxr	Move to Condition Register from XER	X	31	512
mfcrr	Move from Condition Register	X	31	19
mffs[.]	Move from FPSCR	X	63	583
mfmsr	Move from Machine State Register	X	31	83

Item	Description	Description	Description	Description
mfspr	Move from Special-Purpose Register	X	31	339
mfsr	Move from Segment Register	X	31	595
mfsrin	Move from Segment Register Indirect	X	31	659
mtcrf	Move to Condition Register Fields	XFX	31	144
mtfsb0[.]	Move to FPSCR Bit 0	X	63	70
mtfsb1[.]	Move to FPSCR Bit 1	X	63	38
mtfsf[.]	Move to FPSCR Fields	XFL	63	711
mtfsfi[.]	Move to FPSCR Field Immediate	X	63	134
mtmsr	Move to Machine State Register	X	31	146
mtspr	Move to Special-Purpose Register	X	31	467
mtsr	Move to Segment Register	X	31	210
mtsri	Move to Segment Register Indirect	X	31	242
mtsrin	Move to Segment Register Indirect	X	31	242
mul[o][.]	Multiply	XO	31	107
mulhw[.]	Multiply High Word	XO	31	75
mulhwu[.]	Multiply High Word Unsigned	XO	31	11
muli	Multiply Immediate	D	07	
mulli	Multiply Low Immediate	D	07	
mullw[o][.]	Multiply Low Word	XO	31	235
muls[o][.]	Multiply Short	XO	31	235
nabs[o][.]	Negative Absolute	XO	31	488
nand[.]	NAND	X	31	476
neg[o][.]	Negate	XO	31	104
nor[.]	NOR	X	31	124
or[.]	OR	X	31	444
orc[.]	OR with Complement	X	31	412
ori	OR Immediate	D	24	
oril	OR Immediate Lower	D	24	
oris	OR Immediate Shifted	D	25	
oriu	OR Immediate Upper	D	25	
rfi	Return from Interrupt	X	19	50
rlimi[.]	Rotate Left Immediate then Mask Insert	M	20	
rlinm[.]	Rotate Left Immediate then AND with Mask	M	21	
rlmi[.]	Rotate Left then Mask Insert	M	22	
rlnm[.]	Rotate Left then AND with Mask	M	23	

Item	Description	Description	Description	Description
rlwimi[.]	Rotate Left Word Immediate then Mask Insert	M	20	
rlwinm[.]	Rotate Left Word Immediate then AND with Mask	M	21	
rlwnm[.]	Rotate Left Word then AND with Mask	M	23	
rrib[.]	Rotate Right and Insert Bit	X	31	537
sc	System Call	SC	17	
sf[o][.]	Subtract from	XO	31	08
sfe[o][.]	Subtract from Extended	XO	31	136
sfi	Subtract from Immediate	D	08	
sfme[o][.]	Subtract from Minus One Extended	XO	31	232
sfze[o][.]	Subtract from Zero Extended	XO	31	200
si	Subtract Immediate	D	12	
si.	Subtract Immediate and Record	D	13	
sl[.]	Shift Left	X	31	24
sle[.]	Shift Left Extended	X	31	153
sleq[.]	Shift Left Extended with MQ	X	31	217
sliq[.]	Shift Left Immediate with MQ	X	31	184
slliq[.]	Shift Left Long Immediate with MQ	X	31	248
sllq[.]	Shift Left Long with MQ	X	31	216
slq[.]	Shift Left with MQ	X	31	152
slw[.]	Shift Left Word	X	31	24
sr[.]	Shift Right	X	31	536
sra[.]	Shift Right Algebraic	X	31	792
srai[.]	Shift Right Algebraic Immediate	X	31	824
sraiq[.]	Shift Right Algebraic Immediate with MQ	X	31	952
sraq[.]	Shift Right Algebraic with MQ	X	31	920
sraw[.]	Shift Right Algebraic Word	X	31	792
srawi[.]	Shift Right Algebraic Word Immediate	X	31	824
sre[.]	Shift Right Extended	X	31	665
srea[.]	Shift Right Extended Algebraic	X	31	921
sreq[.]	Shift Right Extended with MQ	X	31	729
sriq[.]	Shift Right Immediate with MQ	X	31	696

Item	Description	Description	Description	Description
srlq[.]	Shift Right Long Immediate with MQ	X	31	760
srlq[.]	Shift Right Long with MQ	X	31	728
srq[.]	Shift Right with MQ	X	31	664
srw[.]	Shift Right Word	X	31	536
st	Store	D	36	
stb	Store Byte	D	38	
stbrx	Store Byte-Reversed Indexed	X	31	662
stbu	Store Byte with Update	D	39	
stbux	Store Byte with Update Indexed	X	31	247
stbx	Store Byte Indexed	X	31	215
stfd	Store Floating-Point Double	D	54	
stfdu	Store Floating-Point Double with Update	D	55	
stfdux	Store Floating-Point Double with Update Indexed	X	31	759
stfdx	Store Floating-Point Double Indexed	X	31	727
stfs	Store Floating-Point Single	D	52	
stfsu	Store Floating-Point Single with Update	D	53	
stfsux	Store Floating-Point Single with Update Indexed	X	31	695
stfsx	Store Floating-Point Single Indexed	X	31	663
sth	Store Half	D	44	
sthbrx	Store Half Byte-Reverse Indexed	X	31	918
sthu	Store Half with Update	D	45	
sthux	Store Half with Update Indexed	X	31	439
sthx	Store Half Indexed	X	31	407
stm	Store Multiple	D	47	
stmw	Store Multiple Word	D	47	
stsi	Store String Immediate	X	31	725
stswi	Store String Word Immediate	X	31	725
stswx	Store String Word Indexed	X	31	661
stsx	Store String Indexed	X	31	661
stu	Store with Update	D	37	
stux	Store with Update Indexed	X	31	183
stw	Store	D	36	

Item	Description	Description	Description	Description
stwbrx	Store Word Byte-Reversed Indexed	X	31	662
stwcx.	Store Word Conditional Indexed	X	31	150
stwu	Store Word with Update	D	37	
stwux	Store Word with Update Indexed	X	31	183
stwx	Store Word Indexed	X	31	151
stx	Store Indexed	X	31	151
subf[o][.]	Subtract from	XO	31	40
subfc[o][.]	Subtract from Carrying	XO	31	08
subfe[o][.]	Subtract from Extended	XO	31	136
subfic	Subtract from Immediate Carrying	D	08	
subfme[o][.]	Subtract from Minus One Extended	XO	31	232
subfze[o][.]	Subtract from Zero Extended	XO	31	200
sync	Synchronize	X	31	598
t	Trap	X	31	04
ti	Trap Immediate	D	03	
tlbie	Translation Look-aside Buffer Invalidate Entry	X	31	306
tw	Trap Word	X	31	04
twi	Trap Word Immediate	D	03	
xor[.]	XOR	X	31	316
xori	XOR Immediate	D	26	
xoril	XOR Immediate Lower	D	26	
xoris	XOR Immediate Shift	D	27	
xoriu	XOR Immediate Upper	D	27	

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.

Value	Definition
/, //, ///	Reserved/unused; nominally zero (0).
A	Pseudonym for RA in some diagrams.
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.
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.
MB	Specifies the begin value (bit number) of the mask for the operation.
ME	Specifies the end value (bit number) of the mask for the operation.
NB	Specifies the byte count for the operation.
OE	Specifies that the overflow bits in the Fixed-Point Exception register are affected if the operation results in overflow
RA	Specifies the source general-purpose register for the operation.
RB	Specifies the source general-purpose register for the operation.
RS	Specifies the source general-purpose register for the operation.
RT	Specifies the target general-purpose register where the operation is stored.
S	Pseudonym for RS in some diagrams.
SA	Documented in the svc instruction.
SH	Specifies the (immediate) shift value for the operation.
SI	Specifies the 16-bit signed integer for the operation.
SIMM	16-bit two's-complement value which will be sign-extended for comparison.
SPR	Specifies the source special purpose register for the operation.

Value	Definition
SR	Specifies the source segment register for the operation.
ST	Specifies the target segment register for the operation.
TO	Specifies TO bits that are ANDed with compare results.
U	Specifies source immediate value for operation.
UI	Specifies 16-bit unsigned integer for operation.

Bit 31

Bit 31 is the record bit.

Value	Definition
0	Does not update the condition register.
1	Updates the condition register to reflect the result of the operation.

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.

Storage operands and alignment

The vector data types are 16 bytes in size, and must be aligned on a 16-byte (quadword) boundary.

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 41. Data Types

Contents	New C/C++ Type
16 unsigned char	vector unsigned char
16 signed char	vector signed char
16 unsigned char	vector bool char
8 unsigned short	vector unsigned short
8 signed short	vector signed short
8 unsigned short	vector bool short
4 unsigned int	vector unsigned int
4 signed int	vector signed int
4 unsigned int	vector bool int
4 float	vector float

Register usage conventions

The PowerPC vector extension architecture adds 32 vector registers (VRs).

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 42. VR Conventions

Register	Status	Use
VR0	Volatile	Scratch register.
VR1	Volatile	Scratch register.
VR2	Volatile	First vector argument. First vector of function return value.
VR3	Volatile	Second vector argument, scratch.
VR4	Volatile	Third vector argument, scratch.
VR5	Volatile	Fourth vector argument, scratch.
VR6	Volatile	Fifth vector argument, scratch.
VR7	Volatile	Sixth vector argument, scratch.
VR8	Volatile	Seventh vector argument, scratch.
VR9	Volatile	Eighth vector argument, scratch.
VR10	Volatile	Ninth vector argument, scratch.
VR11	Volatile	Tenth vector argument, scratch.
VR12	Volatile	Eleventh vector argument, scratch.
VR13	Volatile	Twelfth vector argument, scratch.
VR14:19	Volatile	Scratch.
VR20:31	Reserved (default mode) Non-Volatile (extended ABI mode)	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.
VRSAGE	Reserved	In the AIX® ABI, VRSAGE is not used. An ABI-compliant program must not use or alter VRSAGE.
VSCR	Volatile	Vector status and control register. Contains saturation status bit and other than Java™ mode control bit.

The Altivec Programming Interface Specification defines the VRSAGE register to be used as a bitmask of vector registers in use. AIX® requires that an application never modify the VRSAGE register.

Runtime stack

The runtime stack begins quadword aligned for both 32-bit and 64-bit processes.

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.

VRSAGE is not recognized by the AIX® ABI, and should not be used or altered by ABI-compliant programs. The VRSAGE 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 43. Example of a 32-bit Runtime Stack

Item	Description
Sp ->	Back chain
	FPR31 (if needed)
	...
-nFPRs*8	...
	GPR31 (if needed)
	...
-nGPRs*4	...
	VRSAVE
	Alignment padding (to 16-byte boundary)
	VR31 (if needed)
-nVRs*16	...
-220(max)	...
	Local variables
NF+24	Parameter List Area
NF+20	Saved TOC
NF+16	Reserved (binder)
NF+12	Reserved (compiler)
NF+8	Saved LR
NF+4	Saved CR
NF ->	Sp (after NF-newframe allocated)

The 64-bit runtime stack looks like the following (pre-prolog):

Table 44. Example of a 64-bit Runtime Stack

Item	Description
Sp ->	Back chain
	FPR31 (if needed)
	...
-nFPRs*8	...
	GPR31 (if needed)
	...
-nGPRs*8	...
	VRSAVE
	Alignment padding (to 16-byte boundary)
	VR31 (if needed)
-nVRs*16	...
-288(max)	...
	Local variables
NF+48	Parameter List Area
NF+40	Saved TOC
NF+32	Reserved (binder)
NF+24	Reserved (compiler)
NF+16	Saved LR
NF+8	Saved CR

Table 44. Example of a 64-bit Runtime Stack (continued)

Item	Description
NF ->	Sp (after NF-newframe allocated)

Related concepts:

“Assembling and linking a program” on page 73

The assembly language program defines the commands for assembling and linking a program.

“Interpreting an assembler listing” on page 77

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

“Interpreting a symbol cross-reference” on page 82

The example of the symbol cross-reference for the **hello.s** assembly program.

“Subroutine linkage convention” on page 83

The Subroutine Linkage Convention.

“Understanding and programming the toc” on page 110

The TOC is used to find objects in an XCOFF file.

“Running a program” on page 118

A program is ready to run when it has been assembled and linked without producing any error messages.

Vector register save and restore procedures

The vector save and restore functions are provided by the system (libc) as an aid to language compilers.

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.

Item	Description				
_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	

Item	Description				
	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	
	lvx	v23,	r12,	r0	# restore v23
_restv24:	addi	r12,	r0,	-128	
	lvx	v24,	r12,	r0	# restore v24
_restv25:	addi	r12,	r0,	-112	
	lvx	v25,	r12,	r0	# restore v25
_restv26:	addi	r12,	r0,	-96	
	lvx	v26,	r12,	r0	# restore v26
_restv27:	addi	r12,	r0,	-80	
	lvx	v27,	r12,	r0	# restore v27
_restv28:	addi	r12,	r0,	-64	
	lvx	v28,	r12,	r0	# restore v28
_restv29:	addi	r12,	r0,	-48	
	lvx	v29,	r12,	r0	# restore v29
_restv30:	addi	r12,	r0,	-32	
	lvx	v30,	r12,	r0	# restore v30
_restv31:	addi	r12,	r0,	-16	
	lvx	v31,	r12,	r0	# restore v31
	br				

Procedure calling sequence

The procedure calling conventions with respect to argument passing and return values.

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.

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 45. 32-bit Variable Length Argument List Parameters (post-prolog)

Item	Description	
OldSp ->	Back chain (bc)	
-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 46. 64-bit variable length argument list parameters (post-prolog)

Item	Description	
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

The function that returns a vector type or has vector parameters require a function prototype.

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 provides the information necessary to determine the presence of vector state in the stack frame for a function.

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:

Item	Description
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.

Item	Description
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 the function has a variable length argument list */
unsigned vectorparms:7;	/* number of vector parameters if not variable */ /* argument list. Otherwise the mandatory field*/ /* parmsonstk 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:	If vectorparms is non-zero, then the parminfo field is interpreted as: /* b '00' = fixed parameter */ /* b '01' = vector parameter */ /* b '10' = single-precision float parameter */ /* b '11' = double-precision float parameter */

Debug stabstrings

The stabstring codes are defined to specify the location of objects in VRs.

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

The 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.

Notices

This information was developed for products and services offered in the US.

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 grant you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US*

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

*Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan*

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 jurisdictions 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.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide 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 Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US*

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.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

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.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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 actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate 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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year).

Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_.

Privacy policy considerations

IBM® Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as the customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at <http://www.ibm.com/privacy> and IBM’s Online Privacy Statement at <http://www.ibm.com/privacy/details> the section entitled “Cookies, Web Beacons and Other Technologies” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at <http://www.ibm.com/software/info/product-privacy>.

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at www.ibm.com/legal/copytrade.shtml.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Index

Special characters

- .align pseudo-op 517
- .bb pseudo-op 518
- .bc pseudo-op 518
- .bf pseudo-op 519
- .bi pseudo-op 520
- .bs pseudo-op 520
- .byte pseudo-op 521
- .comm pseudo-op 522
- .csect pseudo-op 525
- .double pseudo-op 527
- .drop pseudo-op 528
- .dsect pseudo-op 529
- .eb pseudo-op 531
- .ec pseudo-op 532
- .ef pseudo-op 532
- .ei pseudo-op 533
- .es pseudo-op 534
- .extern pseudo-op 535
- .file pseudo-op 536
- .float pseudo-op 537
- .function pseudo-op 537
- .globl pseudo-op 538
- .hash pseudo-op 539
- .lcomm pseudo-op 541
- .lglobl pseudo-op 542
- .line pseudo-op 543
- .long pseudo-op 544
- .long pseudo-op 544
- .machine pseudo-op 545
- .org pseudo-op 548
- .quad pseudo-op 549
- .rename pseudo-op 46, 550
- .set pseudo-op 551
- .short pseudo-op 552
- .source pseudo-op 553
- .space pseudo-op 554
- .stabx pseudo-op 555
- .string pseudo-op 556
- .tbttag pseudo-op 556
- .tc pseudo-op 558
- .toc pseudo-op 559
- .tocof pseudo-op 560
- .using pseudo-op 561
- .vbyte pseudo-op 565
- .weak pseudo-op 566
- .xline pseudo-op 567

Numerics

- 32-bit applications
 - POWER® family 10
 - PowerPC® 10
- 32-bit fixed-point rotate and shift instructions
 - extended mnemonics 137
- 64-bit fixed-point rotate and shift instructions
 - extended mnemonics 140

A

- a (Add) instruction 158
- abs (Absolute) instruction 154
- accessing data through the TOC 111
- add (Add) instruction 156
- addc (Add Carrying) instruction 158
- adde (Add Extended) instruction 160
- addi (Add Immediate) instruction 162
- addic (Add Immediate Carrying) instruction 163
- addic. (Add Immediate Carrying and Record) instruction 164
- addis (Add Immediate Shifted) instruction 165
- addme (Add to Minus One Extended) instruction 167
- address location
 - making a translation look-aside buffer for
 - using tlbi (Translation Look-Aside Buffer Invalidate Entry) instruction 502
 - using tlbie (Translation Look-Aside Buffer Invalidate Entry) instruction 502
- addresses
 - adding two general-purpose registers
 - using add (Add) instruction 156
 - using cax (Compute Address) instruction 156
 - calculating from an offset less than 32KB
 - using addi (Add Immediate) instruction 162
 - using cal (Compute Address Lower) instruction 162
 - calculating from an offset more than 32KB
 - using addis (Add Immediate Shifted) instruction 165
 - using cau (Compute Address Upper) instruction 165
- addressing
 - absolute 68
 - absolute immediate 68
 - explicit-based 70
 - implicit-based 71
 - location counter 72
 - pseudo-ops 514
 - relative immediate 69
- addze (Add to Zero Extended) instruction 169
- ae (Add Extended) instruction 160
- ai (Add Immediate) instruction 163
- ai. (Add Immediate and Record) instruction 164
- alias
 - creating for an illegal name in syntax
 - using .rename pseudo-op 550
- ame (Add to Minus One Extended) instruction 167
- and (AND) instruction 171
- andc (AND with Complement) instruction 172
- andi. (AND Immediate) instruction 174
- andil. (AND Immediate Lower) instruction 174
- andis. (AND Immediate Shifted) instruction 175
- andiu. (AND Immediate Upper) instruction 175
- Appendix H: Value Definitions 632
- architecture
 - multiple hardware support 2
 - POWER and PowerPC® 10
- as command 73
- assembler
 - features 1
 - interpreting a listing 77
 - passes 75
- assembling
 - program 73

- assembling (*continued*)
 - with the cc command 74
- aze (Add to Zero Extended) instruction 169

B

- b (Branch) instruction 176
- base address
 - specifying
 - using .using pseudo-op 561
- base register
 - assigning a number for
 - using .using pseudo-op 561
 - stop using specified register as
 - using .drop pseudo-op 528
- bbf[l][a] extended mnemonic 120
- bbfc[l] extended mnemonic 120
- bbfr[l] extended mnemonic 120
- bbt[l][a] extended mnemonic 120
- bbtc[l] extended mnemonic 120
- bbtr[l] extended mnemonic 120
- bc (Branch Conditional) instruction 177
- bcc (Branch Conditional to Count Register) instruction 179
- bctr (Branch Conditional to Count Register) instruction 179
- bclr (Branch Conditional Register) instruction 182
- bcr (Branch Conditional Register) instruction 182
- bctr[l] extended mnemonic 120
- bdn[l][a] extended mnemonic 120
- bdnr[l] extended mnemonic 120
- bdnz[l][a] extended mnemonic 120
- bdz[l][a] extended mnemonic 120
- bdzlr[l] extended mnemonic 120
- bdzr[l] extended mnemonic 120
- bf[l][a] extended mnemonic 120
- bl (Branch and Link) instruction 176
- br[l] extended mnemonic 120
- branch instructions
 - extended mnemonics of 119
- branch prediction
 - extended mnemonics for 124
- bt[l][a] extended mnemonic 120

C

- caches
 - using clcs (Cache Line Compute Size) instruction 184
 - using clf (Cache Line Flush) instruction 186
 - using cli (Cache Line Invalidate) instruction 187
 - using dcbf (Data Cache Block Flush) instruction 205
 - using dcbi (Data Cache Block Invalidate) instruction 206
 - using dcbst (Data Cache Block Store) instruction 208
 - using dcbt (Data Cache Block Touch) instruction 209
 - using dcbtst (Data Cache Block Touch for Store) instruction 212
 - using dcbz (Data Cache Block Set to Zero) instruction 214
 - using dclst (Data Cache Line Store) instruction 216
 - using dclz (Data Cache Line Set to Zero) instruction 214
 - using dcs (Data Cache Synchronize) instruction 498
 - using icbi (Instruction Cache Block Invalidate) instruction 282
 - using ics (Instruction Cache Synchronize) instruction 283
- cal (Compute Address Lower) instruction 162
- called routines 104
- calling conventions
 - support for
 - pseudo-ops 515

- calling routines 103
- cau (Compute Address Upper) instruction 165
- cax (Compute Address) instruction 156
- cc command
 - assembling and linking with 74
- character set 32
- character values
 - assembling into consecutive bytes
 - using .string pseudo-op 556
- clcs (Cache Line Compute Size) instruction 184
- clf (Cache Line Flush) instruction 186
- cli (Cache Line Invalidate) instruction 187
- clrldi extended mnemonic 141
- clrslldi extended mnemonic 141
- clrlwi extended mnemonic 139
- clrrdi extended mnemonic 141
- clrrwi extended mnemonic 139
- clrslwi extended mnemonic 139
- cmp (Compare) instruction 189
- cmpi (Compare Immediate) instruction 191
- cmpl (Compare Logical) instruction 192
- cmpli (Compare Logical Immediate) instruction 193
- cntlz (Count Leading Zeros) instruction 196
- cntlzd (Count Leading Zeros Double Word) Instruction 195
- cntlzw (Count Leading Zeros Word) instruction 196
- common blocks
 - defining
 - using .comm pseudo-op 522
 - identifying the beginning of
 - using .bc pseudo-op 518
 - identifying the end of
 - using .ec pseudo-op 532
- Condition Register 201
 - copying bit 3 from the Fixed-Point Exception Register into
 - using mcrxr (Move to Condition Register from XER) instruction 341
 - copying general-purpose register contents into
 - using mtrcf (Move to Condition Register Fields) instruction 351
 - copying Summary Overflow bit from the Fixed-Point Exception Register into
 - using mcrxr (Move to Condition Register from XER) instruction 341
 - copying the Carry bit from the Fixed-Point Exception Register into
 - using mcrxr (Move to Condition Register from XER) instruction 341
 - copying the Overflow bit from the Fixed-Point Exception Register into
 - using mcrxr (Move to Condition Register from XER) instruction 341
- Condition Register bit 197
 - placing ANDing and the complement in a Condition Register bit
 - using crandc (Condition Register AND with Complement) instruction 198
 - placing complemented result of ANDing two Condition Register bits in
 - using crnand (Condition Register NAND) instruction 200
 - placing complemented result of XORing two Condition Register bits in
 - using creqv (Condition Register Equivalent) instruction 199

- Condition Register bit (*continued*)
 - placing result of ORing and complement of Condition Register bit in
 - using crorc (Condition Register OR with Complement) instruction 203
 - placing result of ORing two Condition Register bits in
 - using cror (Condition Register OR) instruction 202
 - placing result of XORing two Condition Register bits in
 - using crxor (Condition Register XOR) instruction 204
- Condition Register field
 - copying the contents from one into another
 - using mcrf (Move Condition Register Field) Instruction 339
- condition register logical instructions
 - extended mnemonics 126
- constants 47
- control sections
 - giving a storage class to
 - using .csect pseudo-op 525
 - giving an alignment to
 - using .csect pseudo-op 525
 - grouping code into
 - using .csect pseudo-op 525
 - grouping data into
 - using .csect pseudo-op 525
 - naming
 - using .csect pseudo-op 525
- count number of one bits in doubleword 383
- Count Register
 - branching conditionally to address in
 - using bcc (Branch Conditional to Count Register) instruction 179
 - using bcctr (Branch Conditional to Count Register) instruction 179
- CPU ID
 - determination 4
- crand (Condition Register AND) instruction 197
- crandc (Condition Register AND with Complement) instruction 198
- crclr extended mnemonic 126
- creqv (Condition Register Equivalent) instruction 199
- crmove extended mnemonic 126
- crnand (Condition Register NAND) instruction 200
- crnor (Condition Register) instruction 201
- crnot extended mnemonic 126
- cror (Condition Register OR) instruction 202
- crorc (Condition Register OR with Complement) instruction 203
- cross-reference
 - interpreting a symbol 82
 - mnemonics 3
- crset extended mnemonic 126
- crxor (Condition Register XOR) instruction 204

D

- data
 - accessing through the TOC 111
- data alignment
 - pseudo-ops 513
- data definition
 - pseudo-ops 513
- dcbf (Data Cache Block Flush) instruction 205
- dcbi (Data Cache Block Invalidate) instruction 206
- dcbst (Data Cache Block Store) instruction 208
- dcbt (Data Cache Block Touch) instruction 209
- dcbtst (Data Cache Block Touch for Store) instruction 212

- dcbz (Data Cache Block Set to Zero) instruction 214
- dclst (Data Cache Line Store) instruction 216
- dclz (Data Cache Line Set to Zero) instruction 214
- dcs (Data Cache Synchronize) instruction 498
- debug traceback tags
 - defining
 - using .tbttag pseudo-op 556
- debuggers
 - providing information to
 - using .stabx pseudo-op 555
 - symbol table entries
 - pseudo-ops 515
- defining
 - table of contents
 - using .tocof pseudo-op 559
- div (Divide) instruction 217
- divd (Divide Double Word) Instruction 219
- divdu (Divide Double Word Unsigned) Instruction 220
- divs (Divide Short) instruction 222
- divw (Divide Word) instruction 224
- divwu (Divide Word Unsigned) instruction 225
- double floating-point constant
 - storing at the next fullword location
 - using .double pseudo-op 527
- double-precision floating-point
 - adding 64-bit operand to result of multiplying two operands
 - using fma (Floating Multiply-Add) instruction 253, 275
 - using fmadd (Floating Multiply-Add) instruction 253, 275
 - adding two 64-bit operands
 - using fa (Floating Add) instruction 240
 - using fadd (Floating Add Double) instruction 240
 - dividing 64-bit operands
 - using fd (Floating Divide) instruction 251
 - using fdiv (Floating Divide Double) instruction 251
 - multiplying two 64-bit operands
 - using fm (Floating Multiply) instruction 259
 - using fmul (Floating Multiply Double) instruction 259
 - multiplying two 64-bit operands and adding to 64-bit operand
 - using fnma (Floating Negative Multiply-Add) instruction 264
 - using fnmadd (Floating Negative Multiply-Add Double) instruction 264
 - multiplying two 64-bit operands and subtracting 64-bit operand
 - using fnms (Floating Negative Multiply-Subtract) instruction 267
 - using fnmsub (Floating Negative Multiply-Subtract Double) instruction 267
 - rounding 64-bit operand to single precision
 - using frsp (Floating Round to Single Precision) instruction 271
 - subtracting 64-bit operand from result of multiplying two 64-bit operands
 - using fms (Floating Multiply-Subtract) instruction 257
 - using fmsub (Floating Multiply-Subtract Double) instruction 257
 - subtracting 64-bit operands
 - using fs (Floating Subtract) instruction 279
 - using fsub (Floating Subtract Double) instruction 279
- doz (Difference or Zero) instruction 227
- dozi (Difference or Zero Immediate) instruction 229
- dummy control sections
 - identifying the beginning of
 - using .dsect pseudo-op 529

dummy control sections (*continued*)
 identifying the continuation of
 using .dsect pseudo-op 529

E

eciwx (External Control In Word Indexed) instruction 230
ecowx (External Control Out Word Indexed) instruction 231
eieio (Enforce In-Order Execution of I/O) instruction 232
epilogs 95
 actions 95
eqv (Equivalent) instruction 234
error
 messages 568
error conditions
 detection 6
expressions 57
 assembling into a TOC entry
 using .tc pseudo-op 558
 assembling into consecutive bytes 521
 assembling into consecutive doublewords
 using .llong pseudo-op 544
 assembling into consecutive fullwords
 using .long pseudo-op 544
 assembling into consecutive halfwords
 using .short pseudo-op 552
 assembling the value into consecutive bytes
 using .vbyte pseudo-op 565
 facilitating the use of local symbols in
 using .tocof pseudo-op 560
 setting a symbol equal in type and value to
 using .set pseudo-op 551
extended mnemonics
 for branch prediction 124
 of 32-bit fixed-point rotate and shift instructions 137
 of 64-bit fixed-point rotate and shift instructions 140
 of branch instructions 119
 of condition register logical instructions 126
 of fixed-point arithmetic instructions 127
 of fixed-point load instructions 129
 of fixed-point logical instructions 130
 of fixed-point trap instructions 131
 of moving from or to special-purpose registers 132
external symbol definitions
 pseudo-ops 514
extldi extended mnemonic 141
extlwi extended mnemonic 139
extrdi extended mnemonic 141
extrwi extended mnemonic 139
exts (Extend Sign) instruction 237
extsb (Extend Sign Byte) instruction 236
extsh (Extend Sign Halfword) instruction 237
extsw (Extend Sign Word) Instruction 233

F

fa (Floating Add) instruction 240
fabs (Floating Absolute Value) instruction 238
fadd (Floating Add Double) instruction 240
fadds (Floating Add Single) instruction 240
fcfid (Floating Convert from Integer Double Word)
 Instruction 242
fcir (Floating Convert Double to Integer with Round)
 instruction 247
fcirz (Floating Convert Double to Integer with Round to Zero)
 instruction 249

fcmpo (Floating Compare Ordered) instruction 243
fcmпу (Floating Compare Unordered) instruction 244
fctid (Floating Convert to Integer Double Word)
 Instruction 245
fctidz (Floating Convert to Integer Double Word with Round
 toward Zero) Instruction 246
fctiw (Floating Convert to Integer Word) instruction 247
fctiwz (Floating Convert to Integer Word with Round to Zero)
 instruction 249
fd (Floating Divide) instruction 251
fdiv (Floating Divide Double) instruction 251
fdivs (Floating Divide Single) instruction 251
fixed-point arithmetic instructions
 extended mnemonics 127
fixed-point load instructions 129
fixed-point logical instructions
 extended mnemonics 130
fixed-point trap instructions 131
floating-point constants
 storing at the next fullword location
 using .float pseudo-op 537
floating-point numbers 29
floating-point registers
 calculating a square root
 using fsqrt (Floating Square Root) instruction 269, 273
 comparing contents of two
 using fcmpo (Floating Compare Ordered)
 instruction 243
 using fcmпу (Floating Compare Unordered)
 instruction 244
 converting 64-bit double-precision floating-point operand
 using fcir (Floating Convert to Integer with Round)
 instruction 247
 using fcirz (Floating Convert Double to Integer with
 Round to Zero) instruction 249
 using fctiw (Floating Convert to Integer Word)
 instruction 247
 using fctiwz (Floating Convert to Integer Word with
 Round to Zero) instruction 249
 converting contents to single precision
 stfsx (Store Floating-Point Single Indexed)
 instruction 466
 using stfs (Store Floating-Point Single) instruction 462
 using stfsu (Store Floating-Point Single with Update)
 instruction 463
 using stfsux (Store Floating-Point Single with Update
 Indexed) instruction 464
 copying contents into Floating-Point Status and Control
 Register
 using mtfsf (Move to FPSCR Fields) instruction 355
 interpreting the contents of 29
 loading converted double-precision floating-point number
 into
 using lfs (Load Floating-Point Single) instruction 304
 using lfsu (Load Floating-Point Single with Update)
 instruction 305
 using lfsux (Load Floating-Point Single with Update
 Indexed) instruction 306
 loading doubleword of data from memory into
 using lfd (Load Floating-Point Double) instruction 294
 using lfdu (Load Floating-Point Double with Update)
 instruction 295
 using lfdux (Load Floating-Point Double with Update
 Indexed) instruction 296
 using lfdx (Load Floating-Point Double Indexed)
 instruction 298

floating-point registers (*continued*)

- loading quadword of data from memory into
 - using `lfq` (Load Floating-Point Quad) instruction 299
 - using `lfqu` (Load Floating-Point Quad with Update) instruction 300
 - using `lfqux` (Load Floating-Point Quad with Update Indexed) instruction 301
 - using `lfqx` (Load Floating-Point Quad Indexed) instruction 303
- moving contents of to another
 - using `fmr` (Floating Move Register) instruction 256
- negating absolute contents of
 - using `fnabs` (Floating Negative Absolute Value) instruction 262
- negating contents of
 - using `fneg` (Floating Negate) instruction 263
- storing absolute value of contents into another
 - using `fabs` (Floating Absolute Value) instruction 238
- storing contents into doubleword storage
 - using `stfd` (Store Floating-Point Double) instruction 453
 - using `stfdu` (Store Floating-Point Double with Update) instruction 454
 - using `stfdx` (Store Floating-Point Double with Update Indexed) instruction 455
 - using `stfdx` (Store Floating-Point Double Indexed) instruction 456
- storing contents into quadword storage
 - using `stfq` (Store Floating-Point Quad) instruction 458
 - using `stfqu` (Store Floating-Point Quad with Update) instruction 459
 - using `stfqux` (Store Floating-Point Quad with Update Indexed) instruction 460
 - using `stfqx` (Store Floating-Point Quad Indexed) instruction 461
- storing contents into word storage
 - using `stfiwx` (Store Floating-Point as Integer word Indexed) instruction 457
- Floating-Point Status and Control Register
 - copying an immediate value into a field of
 - using `mtfsfi` (Move to FPSCR Field Immediate) instruction 357
 - copying the floating-point register contents into
 - using `mtfsf` (Move to FPSCR Fields) instruction 355
 - filling the upper 32 bits after loading
 - using `mffs` (Move from FPSCR) instruction 342
 - loading contents into a floating-point register
 - using `mffs` (Move from FPSCR) instruction 342
 - setting a specified bit to 1
 - using `mtfsb1` (Move to FPSCR Bit 1) instruction 354
 - setting a specified bit to zero
 - using `mtfsb0` (Move to FPSCR Bit 0) instruction 352
- Floating-Point Status and Control Register field
 - copying the bits into the Condition Register
 - using `mcrfs` (Move to Condition Register from FPSCR) instruction 339
- `fm` (Floating Multiply) instruction 259
- `fma` (Floating Multiply-Add) instruction 253
- `fmadd` (Floating Multiply-Add Double) instruction 253
- `fmadds` (Floating Multiply-Add Single) instruction 253
- `fmr` (Floating Move Register) instruction 256
- `fms` (Floating Multiply-Subtract) instruction 257
- `fmsub` (Floating Multiply-Subtract Double) instruction 257
- `fmsubs` (Floating Multiply-Subtract Single) instruction 257
- `fmul` (Floating Multiply) instruction 259
- `fnabs` (Floating Negative Absolute Value) instruction 262
- `fneg` (Floating Negate) instruction 263

- `fnma` (Floating Negative Multiply-Add) instruction 264
- `fnmadd` (Floating Negative Multiply-Add Double) instruction 264
- `fnmadds` (Floating Negative Multiply-Add Single) instruction 264
- `fnms` (Floating Negative Multiply-Subtract) instruction 267
- `fnmsub` (Floating Negative Multiply-Subtract Double) instruction 267
- `fnmsubs` (Floating Negative Multiply-Subtract Single) instruction 267
- `fres` (Floating Reciprocal Estimate Single) instruction 269
- `frsp` (Floating Round to Single Precision) instruction 271
- `frsqre` (Floating Reciprocal Square Root Estimate) instruction 273
- `fs` (Floating Subtract) instruction 279
- `fsel` (Floating-Point Select) instruction 275
- `fsqrt` (Floating Square Root, Double-Precision) instruction 276
- `fsqrts` (Floating Square Root Single) instruction 278
- `fsub` (Floating Subtract Double) instruction 279
- `fsubs` (Floating Subtract Single) instruction 279
- functions
 - identifying
 - using `.function` pseudo-op 537
 - identifying the beginning of
 - using `.bf` pseudo-op 519
 - identifying the end of
 - using `.ef` pseudo-op 532

G

general-purpose registers

- adding complement from -1 with carry
 - using `sfme` (Subtract from Minus One Extended) instruction 492
 - using `subfme` (Subtract from Minus One Extended) instruction 492
- adding contents to the value of the Carry bit
 - using `adde` (Add Extended) instruction 160
 - using `ae` (Add Extended) instruction 160
- adding contents with 16-bit signed integer
 - using `addic` (Add Immediate Carrying) instruction 163
 - using `ai` (Add Immediate) instruction 163
- adding contents with Carry bit and -1
 - using `addme` (Add to Minus One Extended) instruction 167
 - using `ame` (Add to Minus One Extended) instruction 167
- adding immediate value to contents of
 - using `addic` (Add Immediate Carrying and Record) instruction 164
 - using `ai` (Add Immediate and Record) instruction 164
- adding the complement of the contents with the Carry bit
 - using `sfze` (Subtract from Zero Extended) instruction 494
 - using `subfze` (Subtract from Zero Extended) instruction 494
- adding the contents of
 - using `addc` (Add Carrying) instruction 158
- adding zero and the value of the Carry bit to the contents of
 - using `addze` (Add to Zero Extended) instruction 169
 - using `aze` (Add to Zero Extended) instruction 169
- ANDing a generated mask with the rotated contents of
 - using `rlinm` (Rotated Left Immediate Then AND with Mask) instruction 400
 - using `rlhm` (Rotate Left Then AND with Mask) instruction 401

general-purpose registers (*continued*)

- ANDing a generated mask with the rotated contents of (*continued*)
 - using rlwinm (Rotated Left Word Immediate Then AND with Mask) instruction 400
 - using rlwnm (Rotate Left Word Then AND with Mask) instruction 401
- ANDing an immediate value with
 - using andi. (AND Immediate) instruction 174
 - using andil. (AND Immediate Lower) instruction 174
- ANDing contents with the complement of another
 - using andc (AND with Complement) instruction 172
- ANDing logically the contents of
 - using and (AND) instruction 171
- ANDing most significant 16 bits with a 16-bit unsigned integer
 - using andis. (AND Immediate Shifted) instruction 175
 - using andiu. (AND Immediate Upper) instruction 175
- changing the arithmetic sign of the contents of
 - using neg (Negate) instruction 375
- comparing contents algebraically
 - using cmp (Compare) instruction 189
- comparing contents logically
 - using cmpl (Compare Logical) instruction 192
- comparing contents with unsigned integer logically
 - using cmpli (Compare Logical Immediate) instruction 193
- comparing contents with value algebraically
 - using cmpi (Compare Immediate) instruction 191
- computing difference between contents and signed 16-bit integer
 - using dozi (Difference or Zero Immediate) instruction 229
- computing difference between contents of two
 - using doz (Difference or Zero) instruction 227
- copying bit 0 of halfword into remaining 16 bits
 - using lha (Load Half Algebraic) instruction 308
- copying bit 0 of halfword into remaining 16 bits of
 - using lhau (Load Half Algebraic with Update) instruction 309
 - using lhaux (Load Half Algebraic with Update Indexed) instruction 310
 - using lhax (Load Half Algebraic Indexed) instruction 311
- copying Condition Register contents into
 - using mfcrr (Move from Condition Register) instruction 342
 - using mfocrf (Move from One Condition Register Field) instruction 345
 - using mtocrf (Move to One Condition Register Field) instruction 358
- copying contents into a special-purpose register
 - using mtspr (Move to Special-Purpose Register) instruction 359
- copying contents into the Condition Register
 - using mtcrrf (Move to Condition Register Fields) instruction 351
- copying special-purpose register contents into
 - using mfspr (Move from Special-Purpose Register) instruction 346
- copying the Machine State Register contents into
 - using mfmsr (Move from Machine State Register) instruction 344
- dividing by contents of
 - using div (Divide) instruction 217
 - using divs (Divide Short) instruction 222

general-purpose registers (*continued*)

- generating mask of ones and zeros for loading into
 - using maskg (Mask Generate) instruction 336
- inserting contents of one into another under bit-mask control
 - maskir (Mask Insert from Register) instruction 337
- loading consecutive bytes from memory into consecutive
 - using lsi (Load String Immediate) instruction 322
 - using lswi (Load String Word Immediate) instruction 322
 - using lswx (Load String Word Indexed) instruction 324
 - using lsx (Load String Indexed) instruction 324
- loading consecutive bytes into
 - using lscbx (Load String and Compare Byte Indexed) instruction 320
- loading consecutive words into several
 - using lm (Load Multiple) instruction 318
 - using lmw (Load Multiple Word) instruction 318
- loading word of data from memory into
 - using lu (Load with Update) instruction 332
 - using lwzu (Load Word with Zero Update) instruction 332
- loading word of data into
 - using lux (Load with Update Indexed) instruction 333
 - using lwzux (Load Word and Zero with Update Indexed) instruction 333
 - using lwzx (Load Word and Zero Indexed) instruction 334
 - using lx (Load Indexed) instruction 334
- logically complementing the result of ANDing the contents of two
 - using nand (NAND) instruction 374
- logically complementing the result of ORing the contents of two
 - using nor (NOR) instruction 377
- logically ORing the content of two
 - using or (OR) instruction 378
- logically ORing the contents with the complement of the contents of
 - using orc (OR with Complement) instruction 380
- merging a word of zeros with the MQ Register contents
 - using srlq (Shift Right Long with MQ) instruction 439
- merging rotated contents with a word of 32 sign bits
 - using sra (Shift Right Algebraic) instruction 427
 - using srai (Shift Right Algebraic Immediate) instruction 428
 - using sraiQ (Shift Right Algebraic Immediate with MQ) instruction 423
 - using sraq (Shift Right Algebraic with MQ) instruction 425
 - using sraw (Shift Right Algebraic Word) instruction 427
 - using sawi (Shift Right Algebraic Word Immediate) instruction 428
- merging rotated contents with the MQ Register contents
 - using sreq (Shift Right Extended with MQ) instruction 434
 - using srlqi (Shift Right Long Immediate with MQ) instruction 437
 - using srlq (Shift Right Long with MQ) instruction 439
- merging the rotated contents results with the MQ Register contents
 - using sllqi (Shift Left Long Immediate with MQ) instruction 415
- merging with masked MQ Register contents
 - using sleq (Shift Left Extended with MQ) instruction 412

general-purpose registers (*continued*)

- multiplying a word
 - using mulhw (Multiply High Word) instruction 365
 - using mulhwu (Multiply High Word Unsigned) instruction 367
 - multiplying the contents by a 16-bit signed integer
 - using muli (Multiply Immediate) instruction 369
 - using mulli (Multiply Low Immediate) instruction 369
 - multiplying the contents of two
 - using mul (Multiply) instruction 361
 - multiplying the contents of two general-purpose registers into
 - using mullw (Multiply Low Word) instruction 370
 - using muls (Multiply Short) instruction 370
 - negating the absolute value of
 - using nabs (Negative Absolute) instruction 372
 - ORing the lower 16 bits of the contents with a 16-bit unsigned integer
 - using ori (OR Immediate) instruction 381
 - using oril (OR Immediate Lower) instruction 381
 - ORing the upper 16 bits of the contents with a 16-bit unsigned integer
 - using oris (OR Immediate Shifted) instruction 382
 - using oriu (OR Immediate Upper) instruction 382
 - placing a copy of rotated contents in the MQ Register
 - using srea (Shift Right Extended Algebraic) instruction 432
 - placing a copy of rotated data in the MQ register
 - using sle (Shift Left Extended) instruction 410
 - placing number of leading zeros in
 - using cntlz (Count Leading Zeros) instruction 196
 - using cntlzw (Count Leading Zeros Word) instruction 196
 - placing rotated contents in the MQ Register
 - using sliq (Shift Left Immediate with MQ) instruction 413
 - using slq (Shift Left with MQ) instruction 418
 - using sriq (Shift Right Immediate with MQ) instruction 436
 - placing the absolute value of the contents in
 - using abs (Absolute) instruction 154
 - placing the logical AND of the rotated contents in
 - using srq (Shift Right with MQ) instruction 440
 - placing the rotated contents in the MQ register
 - using srq (Shift Right with MQ) instruction 440
 - rotating contents left
 - using rlmi (Rotate Left Then Mask Insert) instruction 396
 - using sl (Shift Left) instruction 419
 - using sle (Shift Left Extended) instruction 410
 - using sliq (Shift Left Immediate with MQ) instruction 413
 - using slliq (Shift Left Long Immediate with MQ) instruction 415
 - using sr (Shift Right) instruction 442
 - using sra (Shift Right Algebraic) instruction 427
 - using sraq (Shift Right Algebraic with MQ) instruction 425
 - using srea (Shift Right Extended Algebraic) instruction 432
 - using sreq (Shift Right Extended with MQ) instruction 434
 - using sriq (Shift Right Immediate with MQ) instruction 436
 - setting remaining 16 bits to 0 after loading
 - using lhz (Load Half and Zero) instruction 314

general-purpose registers (*continued*)

- setting remaining 16 bits to zero after loading
 - using lhzu (Load Half and Zero with Update) instruction 314
 - using lhzx (Load Half and Zero Indexed) instruction 317
 - setting remaining 16 bits to zero in
 - using lhbrx (Load Half Byte-Reverse Indexed) instruction 312
 - using lhzux (Load Half and Zero with Update Indexed) instruction 316
 - storing a byte into memory with the address in
 - using stbu (Store Byte with Update) instruction 444
 - storing a byte of data into memory
 - using stb (Store Byte) instruction 443
 - using stbux (Store Byte with Update Indexed) instruction 445
 - using stbx (Store Byte Indexed) instruction 447
 - storing a byte-reversed word of data into memory
 - using stbrx (Store Byte Reverse Indexed) instruction 478
 - using stwbrx (Store Word Byte Reverse Indexed) instruction 478
 - storing a halfword of data into memory
 - using sth (Store Half) instruction 467
 - using sthu (Store Half with Update) instruction 469
 - using sthux (Store Half with Update Indexed) instruction 470
 - using sthx (Store Half Indexed) instruction 471
 - storing a word of data into memory
 - using st (Store) instruction 477
 - using stu (Store with Update) instruction 481
 - using stux (Store with Update Indexed) instruction 483
 - using stw (Store Word) instruction 477
 - using stwcx (Store Word Conditional Indexed) instruction 480
 - using stwu (Store Word with Update) instruction 481
 - using stwux (Store Word with Update Indexed) instruction 483
 - using stwx (Store Word Indexed) instruction 484
 - using stx (Store Indexed) instruction 484
 - storing consecutive bytes from consecutive registers into memory
 - using stsi (Store String Immediate) instruction 474
 - using stswi (Store String Word Immediate) instruction 474
 - using stswx (Store String Word Indexed) instruction 476
 - using stsx (Store String Indexed) instruction 476
 - storing contents of consecutive registers into memory
 - using stm (Store Multiple) instruction 472
 - using stmw (Store Multiple Word) instruction 472
 - storing halfword of data with 2 bytes reversed into memory
 - using sthbrx (Store Half Byte-Reverse Indexed) instruction 467
 - subtracting contents of one from another
 - using sf (Subtract From) instruction 487
 - using subfc (Subtract from Carrying) instruction 487
 - subtracting from
 - using subf (Subtract From) instruction 485
 - subtracting the contents from a 16-bit signed integer
 - using sfi (Subtract from Immediate) instruction 491
 - using subfc (Subtract from Immediate Carrying) instruction 491
 - subtracting the contents from the sum of
 - using sfe (Subtract from Extended) 489

general-purpose registers (*continued*)

- subtracting the contents from the sum of (*continued*)
 - using `subfe` (Subtract from Extended) 489
- subtracting the value of a signed integer from the contents of
 - using `si` (Subtract Immediate) instruction 407
 - using `si`. (Subtract Immediate and Record) instruction 408
- translate effective address into real address and store in
 - using `rac` (Real Address Compute) instruction 384
- using `a` (Add) instruction 158
- using `divw` (Divide Word) instruction 224
- using `divwu` (Divide Word Unsigned) instruction 225
- using `extsb` (Extend Sign Byte) instruction 236
- using `lfq` (Load Floating-Point Quad) instruction 299
- using `lfqu` (Load Floating-Point Quad with Update) instruction 300
- using `lfqux` (Load Floating-Point Quad with Update Indexed) instruction 301
- using `lfqx` (Load Floating-Point Quad Indexed) instruction 303
- using `lwarx` (Load Word and Reserve Indexed) instruction 327
- using `rlimi` (Rotate Left Immediate Then Mask Insert) instruction 398
- using `rlnm` (Rotate Left Then AND with Mask) instruction 401
- using `rlwimi` (Rotate Left Word Immediate Then Mask Insert) instruction 398
- using `rlwnm` (Rotate Left Word Then AND with Mask) instruction 401
- using `rrib` (Rotate Right and Insert Bit) instruction 404
- using `slq` (Shift Left Long with MQ) instruction 416
- using `slq` (Shift Left with MQ) instruction 418
- using `slw` (Shift Left Word) instruction 419
- using `srai` (Shift Right Algebraic Immediate) instruction 428
- using `srai` (Shift Right Algebraic Immediate with MQ) instruction 423
- using `sraw` (Shift Right Algebraic Word) instruction 427
- using `srawi` (Shift Right Algebraic Word Immediate) instruction 428
- using `sre` (Shift Right Extended) instruction 431
- using `srlq` (Shift Right Long Immediate with MQ) instruction 437
- using `srlq` (Shift Right Long with MQ) instruction 439
- using `srq` (Shift Right with MQ) instruction 440
- using `srw` (Shift Right Word) instruction 442
- using `stfq` (Store Floating-Point Quad) instruction 458
- using `stfqu` (Store Floating-Point Quad with Update) instruction 459
- using `stfqux` (Store Floating-Point Quad with Update Indexed) instruction 460
- using `stfqx` (Store Floating-Point Quad Indexed) instruction 461
- XORing contents of
 - using `eqv` (Equivalent) instruction 234
- XORing the contents and 16-bit unsigned integer
 - using `xori` (XOR Immediate) instruction 510
 - using `xoril` (XOR Immediate Lower) instruction 510
- XORing the contents of two
 - using `xor` (XOR) instruction 509
- XORing the upper 16 bits with a 16-bit unsigned integer
 - using `xoris` (XOR Immediate Shift) instruction 511
 - using `xoriu` (XOR Immediate Upper) instruction 511

H

hash values

- associating with external symbol
 - using `.hash` pseudo-op 539

 host machine independence 2

I

`icbi` (Instruction Cache Block Invalidate) instruction 282

`ics` (Instruction Cache Synchronize) instruction 283

implementation

- multiple platform support 2

included files

- identifying the beginning of
 - using `.bi` pseudo-op 520
- identifying the end of
 - using `.ei` pseudo-op 533

inner blocks

- identifying the beginning of
 - using `.bb` pseudo-op 518
- identifying the end of
 - using `.eb` pseudo-op 531

`inslwi` extended mnemonic 139

`insrdi` extended mnemonic 141

`insrwi` extended mnemonic 139

installing the assembler 9

instruction fields 16

instruction forms 12

instructions

- branch 20
- common to POWER and PowerPC 606
- condition register 21
- fixed-point
 - address computation 24
 - arithmetic 24
 - compare 25
 - load and store 22
 - load and store with update 23
 - logical 26
 - move to or from special-purpose registers 27
 - rotate and shift 26
 - string 23
 - trap 25
- floating-point
 - arithmetic 30
 - compare 31
 - conversion 31
 - load and store 29
 - move 30
 - multiply-add 30
 - status and control register 32

PowerPC 616

PowerPC 601 RISC Microprocessor 624

sorted by mnemonic 585

sorted by primary and extended op code 595

system call 21

intermodule calls using the TOC 114

interrupts

- generating when a condition is true
 - using `t` (Trap) instruction 507
 - using `ti` (Trap Immediate) instruction 508
 - using `tw` (Trap Word) instruction 507
 - using `twi` (Trap Word Immediate) instruction 508

supervisor call

- generating an interrupt 496

- interrupts (*continued*)
 - system call
 - generating an interrupt 405
 - system call vectored
 - generating an interrupt 406
- isync (Instruction Synchronize) instruction 283

L

- l (Load) instruction 331
- la extended mnemonic 129
- lbrx (Load Byte-Reverse Indexed) instruction 330
- lbz (Load Byte and Zero) instruction 284
- lbzux (Load Byte and Zero with Update Indexed) instruction 286
- lbzx (Load Byte and Zero Indexed) instruction 288
- ld (Load Double Word) instruction 289
- ldarx (Load Doubleword Reserve Indexed) Instruction 290
- ldu (Load Doubleword with Update) Instruction 291
- ldux (Load Doubleword with Update Indexed) Instruction 293
- ldx (Load Doubleword Indexed) Instruction 293
- leading zeros
 - placing in a general-purpose register
 - using cntlz (Count Leading Zeros) instruction 196
 - using cntlzw (Count Leading Zeros Word) instruction 196
- lfd (Load Floating-Point Double) instruction 294
- lfdu (Load Floating-Point Double with Update) instruction 295
- lfdux (Load Floating-Point Double with Update Indexed) instruction 296
- lfdx (Load Floating-Point Double Indexed) instruction 298
- lfq (Load Floating-Point Quad) instruction 299
- lfqu (Load Floating-Point Quad with Update) instruction 300
- lfqux (Load Floating-Point Quad with Update Indexed) instruction 301
- lfqx (Load Floating-Point Quad Indexed) instruction 303
- lfs (Loading Floating-Point Single) instruction 304
- lfsu (Load Floating-Point Single with Update) instruction 305
- lfsux (Load Floating-Point Single with Update Indexed) instruction 306
- lfsx (Load Floating-Point Single Indexed) instruction 307
- lha (Load Half Algebraic) instruction 308
- lhau (Load Half Algebraic with Update) instruction 309
- lhaux (Load Half Algebraic with Update Indexed) instruction 310
- lhax (Load Half Algebraic Indexed) instruction 311
- lhrx (Load Half Byte-Reverse Indexed) instruction 312
- lh (Load Half and Zero) instruction 314
- lhzu (Load Half and Zero with Update) instruction 314
- lhzux (Load Half and Zero with Update Indexed) instruction 316
- lhzx (Load Half and Zero Indexed) instruction 317
- li extended mnemonic 129
- lil extended mnemonic 129
- line format 34
- line numbers
 - identifying
 - using .line pseudo-op 543
- lines
 - representing the number of
 - using .xline pseudo-op 567
- Link Register
 - branching conditionally to address in
 - using bclr (Branch Conditional Register) instruction 182

- Link Register (*continued*)
 - branching conditionally to address in (*continued*)
 - using bcr (Branch Conditional Register) instruction 182
- linkage
 - subroutine linkage convention 83
- linker
 - making a symbol globally visible to the
 - using .globl pseudo-op 538
- linking 73
 - with the cc command 74
- lis extended mnemonic 129
- listing
 - interpreting an assembler 77
- liu extended mnemonic 129
- lm (Load Multiple) instruction 318
- lmw (Load Multiple Word) instruction 318
- local common section
 - defining a
 - using .lcomm pseudo-op 541
- local symbol
 - facilitating use in expressions
 - using .tocof pseudo-op 560
- location counter 72
 - advancing until a specified boundary is reached
 - using .align pseudo-op 517
 - setting the value of the current
 - using .org pseudo-op 548
- logical processing
 - model 10
- lq (Load Quad Word) instruction 319
- lscbx (Load String and Compare Byte Indexed) instruction 320
- lsi (Load String Immediate) instruction 322
- lswi (Load String Word Immediate) instruction 322
- lsx (Load String Word Indexed) instruction 324
- lsx (Load String Indexed) instruction 324
- lu (Load with Update) instruction 332
- lux (Load with Update Indexed) instruction 333
- lwa (Load Word Algebraic) Instruction 326
- lwarx (Load Word and Reserve Indexed) instruction 327
- lwaux (Load Word Algebraic with Update Indexed) Instruction 328
- lwax (Load Word Algebraic Indexed) Instruction 329
- lwbrx (Load Word Byte-Reverse Indexed) instruction 330
- lwz (Load Word and Zero) instruction 331
- lwzu (Load Word with Zero Update) instruction 332
- lwzux (Load Word and Zero with Update Indexed) instruction 333
- lwzx (Load Word and Zero Indexed) instruction 334
- lx (Load Indexed) instruction 334

M

- Machine State Register
 - after a supervisor call and reinitialize
 - using rfsvc (Return from SVC) instruction 387
 - continue processing and reinitialize
 - using rfi (Return from Interrupt) instruction 386
 - copying the contents into a general-purpose register
 - using mfmsr (Move from Machine State Register) instruction 344
- main memory
 - ensuring storage access in
 - using eieio (Enforce In-Order Execution of I/O) instruction 232
- maskg (Mask Generate) instruction 336
- maskir (Mask Insert from Register) instruction 337

- masksgenerating instance of ones and zeros
 - using maskg (Mask Generate) instruction 336
- mcrf (Move Condition Register Field) instruction 339
- mcrfs (Move to Condition Register from FPSCR) instruction 339
- mcrxr (Move to Condition Register from XER) instruction 341
- memory
 - loading a byte of data from
 - using lbzu (Load Byte and Zero with Update) instruction 285
 - loading byte of data from
 - using lbz (Load Byte and Zero) instruction 284
 - using lbzux (Load Byte and Zero with Update Indexed) instruction 286
 - loading byte of data into
 - using lbzx (Load Byte and Zero Indexed) instruction 288
 - loading byte-reversed halfword of data from
 - using lbrx (Load Half Byte-Reverse Indexed) instruction 312
 - loading byte-reversed word of data from
 - using lbrx (Load Byte-Reverse Indexed) instruction 330
 - using lwbrx (Load Word Byte-Reverse Indexed) instruction 330
 - loading consecutive bytes from
 - using lsi (Load String Immediate) instruction 322
 - using lswi (Load String Word Immediate) instruction 322
 - using lswx (Load String Word Indexed) instruction 324
 - using lsx (Load String Indexed) instruction 324
 - loading doubleword of data from
 - using lfd (Load Floating-Point Double) instruction 294
 - using lfdx (Load Floating-Point Double with Update Indexed) instruction 295
 - using lfdx (Load Floating-Point Double Indexed) instruction 296
 - using lfdx (Load Floating-Point Double Indexed) instruction 298
 - loading halfword of data from
 - using lha (Load Half Algebraic) instruction 308
 - using lhau (Load Half Algebraic with Update) instruction 309
 - using lhaux (Load Half Algebraic with Update Indexed) instruction 310
 - using lhax (Load Half Algebraic Indexed) instruction 311
 - using lhz (Load Half and Zero) instruction 314
 - using lhzu (Load Half and Zero with Update) instruction 314
 - using lhzux (Load Half and Zero with Update Indexed) instruction 316
 - using lhzx (Load Half and Zero Indexed) instruction 317
 - loading quadword of data from
 - using lfq (Load Floating-Point Quad) instruction 299
 - using lfqu (Load Floating-Point Quad with Update) instruction 300
 - using lfqux (Load Floating-Point Quad with Update Indexed) instruction 301
 - using lfqx (Load Floating-Point Quad Indexed) instruction 303
 - loading single-precision floating-point number from
 - using lfsu (Load Floating-Point Single with Update) instruction 305
 - loading single-precision floating-point number from
 - (continued)
 - loading single-precision floating-point number from
 - using lfsx (Load Floating-Point Single Indexed) instruction 307
 - loading single-precision floating-point number into
 - using lfs (Load Floating-Point Single) instruction 304
 - using lfsux (Load Floating-Point Single with Update Indexed) instruction 306
 - loading word of data from
 - using lu (Load with Update) instruction 332
 - using lux (Load with Update Indexed) instruction 333
 - using lwzu (Load Word with Zero Update) instruction 332
 - using lwzux (Load Word and Zero with Update Indexed) instruction 333
 - using lwzx (Load Word and Zero Indexed) instruction 334
 - using lx (Load Indexed) instruction 334
 - setting remaining 24 bits after loading into
 - using lbzx (Load Byte and Zero Indexed) instruction 288
 - setting remaining 24 bits to 0 after loading from
 - using lbz (Load Byte and Zero) instruction 284
 - using lbzux (Load Byte and Zero with Update Indexed) instruction 286
 - setting remaining 24 bits to 0 after loading into
 - using lbzu (Load Byte and Zero with Update) instruction 285
 - storing a quadword of data into
 - using stfq (Store Floating-Point Quad) instruction 458
 - using stfqu (Store Floating-Point Quad with Update) instruction 459
 - using stfqux (Store Floating-Point Quad with Update Indexed) instruction 460
 - using stfqx (Store Floating-Point Quad Indexed) instruction 461
 - using dcbf (Data Cache Block Flush) instruction 205
 - messages
 - error 568
 - warning 568
 - mcr (Move from Condition Register) instruction 342
 - mfctr extended mnemonic 135
 - mfdar extended mnemonic 135
 - mfdec extended mnemonic 135
 - mfdsisr extended mnemonic 135
 - mfear extended mnemonic 135
 - mffs (Move from FPSCR) instruction 342
 - mflr extended mnemonic 135
 - mfmq extended mnemonic 135
 - mfmsr (Move from Machine State Register) instruction 344
 - mfocrf (Move from One Condition Register Field) instruction 345
 - mfpvr extended mnemonic 135
 - mfrctl extended mnemonic 135
 - mfrtcu extended mnemonic 135
 - mfsdr1 extended mnemonic 135
 - mfspir (Move from Special-Purpose Register) instruction 346
 - mfspg extended mnemonic 135
 - mfsrc (Move from Segment Register) instruction 348
 - mfsrci (Move from Segment Register Indirect) instruction 349
 - mfsrcin (Move from Segment Register Indirect) instruction 350
 - mfsrr0 extended mnemonic 135
 - mfsrr1 extended mnemonic 135
 - mfxxer extended mnemonic 135
 - milicode routines 108

- mnemonic
 - instructions sorted by 585
- mnemonics cross-reference 3
- moving from or to special-purpose registers
 - extended mnemonics 132
- mr (Move Register) instruction 378
- mr[.] extended mnemonic 130
- mtrcf (Move to Condition Register Fields) instruction 351
- mtctr extended mnemonic 136
- mtdar extended mnemonic 136
- mtdec extended mnemonic 136
- mtdsir extended mnemonic 136
- mtear extended mnemonic 136
- mtfsb0 (Move to FPSCR Bit 0) instruction 352
- mtfsb1 (Move to FPSCR Bit 1) instruction 354
- mtfsf (Move to FPSCR Fields) instruction 355
- mtfsfi (Move to FPSCR Field Immediate) instruction 357
- mtlr extended mnemonic 136
- mtmq extended mnemonic 136
- mtocrf (Move to One Condition Register Field)
 - instruction 358
- mtrtcl extended mnemonic 136
- mtrtcu extended mnemonic 136
- mtsdr1 extended mnemonic 136
- mtspr (Move to Special-Purpose Register) instruction 359
- mtsprg extended mnemonic 136
- mtsrr0 extended mnemonic 136
- mtsrr1 extended mnemonic 136
- mtxer extended mnemonic 136
- mul (Multiply) instruction 361
- mulhd (Multiply High Double Word) Instruction 363
- mulhdu (Multiply High Double Word Unsigned)
 - Instruction 364
- mulhw (Multiply High Word) instruction 365
- mulhwu (Multiply High Word Unsigned) instruction 367
- muli (Multiply Immediate) instruction 369
- mulld (Multiply Low Double Word) Instruction 368
- mulldo (Multiply Low Double Word) Instruction 368
- mulli (Multiply Low Immediate) instruction 369
- mullw (Multiply Low Word) instruction 370
- muls (Multiply Short) instruction 370

N

- nabs (Negative Absolute) instruction 372
- name
 - creating a synonym or alias for an illegal name
 - using .rename pseudo-op 550
- nand (NAND) instruction 374
- neg (Negate) instruction 375
- nop extended mnemonic 130
- nor (NOR) instruction 377
- not[.] extended mnemonic 130
- notational conventions
 - pseudo-ops 516

O

- op code
 - instructions sorted by primary and extended 595
- operators 55
- or (OR) instruction 378
- orc (OR with Complement) instruction 380
- ori (OR Immediate) instruction 381
- oril (OR Immediate Lower) instruction 381
- oris (OR Immediate Shifted) instruction 382

- oriu (OR Immediate Upper) instruction 382
- output file
 - skipping a specified number of bytes in
 - using .space pseudo-op 554

P

- passes
 - assembler 75
- popcntbd (Population Count Byte Doubleword) 383
- POWER and POWER2
 - instructions 609
- POWER and PowerPC
 - common instructions 606
- POWER and PowerPC instructions
 - extended mnemonics changes 147
 - functional differences for 144
 - PowerPC instructions 152
 - with same op code 145
- POWER and PowerPC®
 - architecture 10
- PowerPC
 - instructions 616
- PowerPC 601 RISC Microprocessor
 - instructions 624
- PowerPC instructions
 - added 152
- process
 - runtime process stack 91
- program
 - running a 118
- programs
 - generating interrupt
 - using t (Trap) instruction 507
 - using ti (Trap Immediate) instruction 508
 - using tw (Trap Word) instruction 507
 - using twi (Trap Word Immediate) instruction 508
- prologs 95
 - actions 95
- pseudo-ops 46, 513, 514, 516, 517, 518, 519, 520, 521, 522, 525, 527, 528, 529, 531, 532, 533, 534, 535, 536, 537, 538, 539, 541, 542, 543, 544, 545, 548, 549, 550, 551, 552, 553, 554, 555, 556, 558, 559, 560, 561, 565, 566, 567
 - .comment 524
 - .except 534
 - .info 540
 - addressing 514
 - calling conventions
 - support for 515
 - data alignment 513
 - functional groups 513
 - miscellaneous 515
 - support for calling conventions 515
 - symbol table entries for debuggers 515

Q

- quad floating-point constant
 - storing at the next fullword location
 - using .quad pseudo-op 549

R

- rac (Real Address Compute) instruction 384

- real address
 - translating effective address to
 - using `eciwx` (External Control In Word Indexed) instruction 230
 - using `ecowx` (External Control Out Word Indexed) instruction 231
- reciprocal, floating single estimate 269
- reciprocal, floating square root estimate 273
- ref pseudo-op 549
- registers
 - special-purpose
 - changes and field handling 8
 - extended mnemonics 132
 - usage and conventions 84
- relocation specifiers
 - `QualNames` 47
- reserved words 33
- `rfi` (Return from Interrupt) instruction 386
- `rfid` (Return from Interrupt Double Word) Instruction 386
- `rfsvc` (Return from SVC) instruction 387
- `rldcl` (Rotate Left Double Word then Clear Left) Instruction 388
- `rldcr` (Rotate Left Double Word then Clear Right) Instruction 390
- `rldic` (Rotate Left Double Word Immediate then Clear) Instruction 391
- `rldicl` (Rotate Left Double Word Immediate then Clear Left) Instruction 389, 392
- `rldicr` (Rotate Left Double Word Immediate then Clear Right) Instruction 394
- `rldimi` (Rotate Left Double Word Immediate then Mask Insert) Instruction 395
- `rlimi` (Rotate Left Immediate Then Mask Insert) instruction 398
- `rlinm` (Rotate Left Immediate Then AND with Mask) instruction 399
- `rlmi` (Rotate Left Then Mask Insert) instruction 396
- `rlnm` (Rotate Left Then AND with Mask) instruction 401
- `rlwimi` (Rotate Left Word Immediate Then Mask Insert) instruction 398
- `rlwinm` (Rotate Left Word Immediate Then AND with Mask) instruction 399
- `rlwnm` (Rotate Left Word Then AND with Mask) instruction 401
- `rotld` extended mnemonic 141
- `rotldi` extended mnemonic 141
- `rotlw` extended mnemonic 139
- `rotlwi` extended mnemonic 139
- `rotrdi` extended mnemonic 141
- `rotrwi` extended mnemonic 139
- `rrib` (Rotate Right and Insert Bit) instruction 404
- running a program 118

S

- `sc` (System Call) instruction 405
- `scv` (System Call Vectored) instruction 406
- section definition
 - pseudo-ops 514
- Segment Register
 - copying to general-purpose registers
 - using `mfsr` (Move from Segment Register) instruction 348
 - using `mfsri` (Move from Segment Register Indirect) instruction 349
 - using `mfsrin` (Move from Segment Register Indirect) instruction 350

- selecting operand with `fsel` instruction 275
- `sf` (Subtract from) instruction 487
- `sfe` (Subtract from Extended) instruction 489
- `sfi` (Subtract from Immediate) instruction 491
- `sfme` (Subtract from Minus One Extended) instruction 492
- `sfze` (Subtract from Zero Extended) instruction 494
- `si` (Subtract Immediate) instruction 407
- `si.` (Subtract Immediate and Record) instruction 408
- `si[.]` extended mnemonic 127
- signed integers
 - extending 16-bit to 32 bits
 - using `exts` (Extend Sign) instruction 237
 - using `extsh` (Extend Sign Halfword) instruction 237
- single-precision floating-point
 - adding 32-bit operand to result of multiplying two operands
 - using `fmadds` (Floating Multiply-Add Single) instruction 253, 275
 - adding two 32-bit operands
 - using `fadds` (Floating Add Single) instruction 240
 - dividing 32-bit operands
 - using `fdivs` (Floating Divide Single) instruction 251
 - multiplying two 32-bit operands
 - using `fmuls` (Floating Multiply Single) instruction 259
 - multiplying two 32-bit operands and adding to 32-bit operand
 - using `fnmadds` (Floating Negative Multiply-Add Single) instruction 264
 - multiplying two 32-bit operands and subtracting 32-bit operand
 - using `fnmsubs` (Floating Negative Multiply-Subtract Single) instruction 267
 - subtracting 32-bit operand from result of multiplying two 32-bit operands
 - using `fmsubs` (Floating Multiply-Subtract Single) instruction 257
 - subtracting 32-bit operands
 - using `fsubs` (Floating Subtract Single) instruction 279
- `sl` (Shift Left) instruction 419
- `sld` (Shift Left Double Word) Instruction 409
- `sldi` extended mnemonic 141
- `sle` (Shift Left Extended) instruction 410
- `sleq` (Shift Left Extended with MQ) instruction 412
- `sliq` (Shift Left Immediate with MQ) instruction 413
- `slliq` (Shift Left Long Immediate with MQ) instruction 415
- `sllq` (Shift Left Long with MQ) instruction 416
- `slq` (Shift Left with MQ) instruction 418
- `slw` (Shift Left Word) instruction 419
- `slwi` extended mnemonic 139
- source files
 - identifying file names
 - using `.file` pseudo-op 536
- source language type 5
 - identifying
 - using `.source` pseudo-op 553
- source module
 - identifying a symbol defined in another
 - using `.extern` pseudo-op 535
- special-purpose registers
 - changes and field handling 8
 - copying general-purpose register contents into
 - using `mtspr` (Move to Special-Purpose Register) instruction 359
 - copying the contents into a general-purpose register
 - using `mfspr` (Move from Special-Purpose Register) instruction 346
 - extended mnemonics 132

- split-field notation 15
- square root, reciprocal floating estimate 273
- sr (Shift Right) instruction 442
- sra (Shift Right Algebraic) instruction 427
- sradd (Shift Right Algebraic Double Word) Instruction 421
- sradi (Shift Right Algebraic Double Word Immediate) Instruction 422
- srai (Shift Right Algebraic Immediate) instruction 428
- sraiq (Shift Right Algebraic Immediate with MQ) instruction 423
- sraq (Shift Right Algebraic with MQ) instruction 425
- sraw (Shift Right Algebraic Word) instruction 427
- srawi (Shift Right Algebraic Word Immediate) instruction 428
- srd (Shift Right Double Word) Instruction 430
- srdis extended mnemonic 141
- sre (Shift Right Extended) instruction 431
- srea (Shift Right Extended Algebraic) instruction 432
- sreq (Shift Right Extended with MQ) instruction 434
- sriq (Shift Right Immediate with MQ) instruction 436
- srliq (Shift Right Long Immediate with MQ) instruction 437
- srliq (Shift Right Long with MQ) instruction 439
- srq (Shift Right with MQ) instruction 440
- srw (Shift Right Word) instruction 442
- srwi extended mnemonic 139
- st (Store) instruction 477
- stack
 - runtime process 91
 - stack-related system standards 95
- statements 35
- static blocks
 - identifying the beginning of
 - using .bs pseudo-op 520
 - identifying the end of
 - using .es pseudo-op 534
- static name
 - keeping information in the symbol table
 - using .lglobl pseudo-op 542
- stb (Store Byte) instruction 443
- stbrx (Store Byte-Reverse Indexed) instruction 478
- stbu (Store Byte with Update) instruction 444
- stbux (Store Byte with Update Indexed) instruction 445
- stbx (Store Byte Indexed) instruction 447
- std (Store Double Word) Instruction 448
- stdcx. (Store Double Word Conditional Indexed) Instruction 448
- stdu (Store Double Word with Update) Instruction 450
- stdux (Store Double Word with Update Indexed) Instruction 451
- stdx (Store Double Word Indexed) Instruction 452
- stfd (Store Floating-Point Double) instruction 453
- stfdu (Store Floating-Point Double with Update) instruction 454
- stfdx (Store Floating-Point Double with Update Indexed) instruction 455
- stfdx (Store Floating-Point Double Indexed) instruction 456
- stfiwx (Store Floating-Point as Integer Word Indexed) instruction 457
- stfq (Store Floating-Point Quad) instruction 458
- stfqu (Store Floating-Point Quad with Update) instruction 459
- stfqux (Store Floating-Point Quad with Update Indexed) instruction 460
- stfqux (Store Floating-Point Quad Indexed) instruction 461
- stfs (Store Floating-Point Single) instruction 462
- stfsu (Store Floating-Point Single with Update) instruction 463
- stfsux (Store Floating-Point Single with Update Indexed) instruction 464
- stfsx (Store Floating-Point Single Indexed) instruction 466
- sth (Store Half) instruction 467
- sthrx (Store Half Byte-Reverse Indexed) instruction 467
- sthu (Store Half with Update) instruction 469
- sthux (Store Half with Update Indexed) instruction 470
- sthx (Store Half Indexed) instruction 471
- storage
 - synchronize
 - using sync (Synchronize) instruction 498
- storage definition
 - pseudo-ops 514
- storage-mapping classes 525
- store
 - quad word 473
- stq (Store Quad Word) instruction 473
- stsi (Store String Immediate) instruction 474
- stswi (Store String Word Immediate) instruction 474
- stswx (Store String Word Indexed) instruction 476
- stsx (Store String Indexed) instruction 476
- stu (Store with Update) instruction 481
- stux (Store with Update Indexed) instruction 483
- stw (Store Word) instruction 477
- stwbrx (Store Word Byte-Reverse Indexed) instruction 478
- stwcx. (Store Word Conditional Indexed) instruction 480
- stwu (Store Word with Update) instruction 481
- stwux (Store Word with Update Indexed) instruction 483
- stwx (Store Word Indexed) instruction 484
- stx (Store Indexed) instruction 484
- sub[o]. extended mnemonic 127
- subc[o]. extended mnemonic 127
- subf (Subtract From) instruction 485
- subfc (Subtract from Carrying) instruction 487
- subfe (Subtract from Extended) instruction 489
- subfc (Subtract from Immediate Carrying) instruction 491
- subfme (Subtract from Minus One Extended) instruction 492
- subfze (Subtract from Zero Extended) instruction 494
- subi extended mnemonic 127
- subic. extended mnemonic 127
- subis extended mnemonic 127
- subroutine
 - linkage convention 83
- svc (Supervisor Call) instruction 496
- symbol table
 - entries for debuggers
 - pseudo-ops 515
 - keeping information of a static name in the
 - using .lglobl pseudo-op 542
- symbols
 - associating a hash value with external
 - using .hash pseudo-op 539
 - constructing 40
 - interpreting a cross-reference 82
 - making globally visible to linker
 - using .globl pseudo-op 538
 - setting equal to an expression in type and value
 - using .set pseudo-op 551
- sync (Synchronize) instruction 498
- synchronize
 - using isync (Instruction Synchronize) instruction 283
- syntax and semantics
 - character set 32
 - comments 39
 - constants 47
 - constructing symbols 40
 - expressions 57

- syntax and semantics (*continued*)
 - instruction statements 35
 - labels 37
 - line format 34
 - mnemonics 38
 - null statements 36
 - operands 38
 - operators 55
 - pseudo-operation statements 35
 - reserved words 33
 - separator character 36
 - statements 35

T

- t (Trap) instruction 507
- table of contents
 - defining
 - using .toc pseudo-op 559
- tags
 - traceback 105
- target addresses
 - branching conditionally to
 - using bc (Branch Conditional) instruction 177
 - branching to
 - using b (Branch) instruction 176
- target environment
 - defining
 - using .machine pseudo-op 545
 - indicator flag 2
- td (Trap Double Word) Instruction 499
- tdi (Trap Double Word Immediate) Instruction 501
- thread local storage 116
- ti (Trap Immediate) instruction 508
- tlbi (Translation Look-Aside Buffer Invalidate Entry) instruction 502
- tlbie (Translation Look-Aside Buffer Invalidate Entry) instruction 502
- tlbld (Load Data TLB Entry) instruction 503
- tlbli (Load Instruction TLB Entry) instruction 505
- tlbsync (Translation Look-Aside Buffer Synchronize) Instruction 506
- TOC
 - accessing data through 111
 - intermodule calls using 114
 - programming the 110
 - understanding the 110
- traceback tags 105
- tw (Trap Word) instruction 507
- twi (Trap Word Immediate) instruction 508

U

- user register set
 - POWER[®] family 10
 - PowerPC[®] 10
- using .weak pseudo-op 566

V

- variable
 - storage-mapping 116
- Vector Processor 634
 - debug stabstrings 641
 - legacy ABI
 - compatibility 641

- Vector Processor (*continued*)
 - legacy ABI (*continued*)
 - interoperability 641
 - procedure calling sequence 639
 - argument passing 639
 - function return values 640
 - register usage conventions 634
 - run-time stack 635
 - storage operands and alignment 634
 - traceback tables 640
 - vector register
 - save and restore 637

W

- warning messages 8, 568

X

- xor (XOR) instruction 509
- xori (XOR Immediate) instruction 510
- xoril (XOR Immediate Lower) instruction 510
- xoris (XOR Immediate Shift) instruction 511
- xoriu (XOR Immediate Upper) instruction 511



Printed in USA