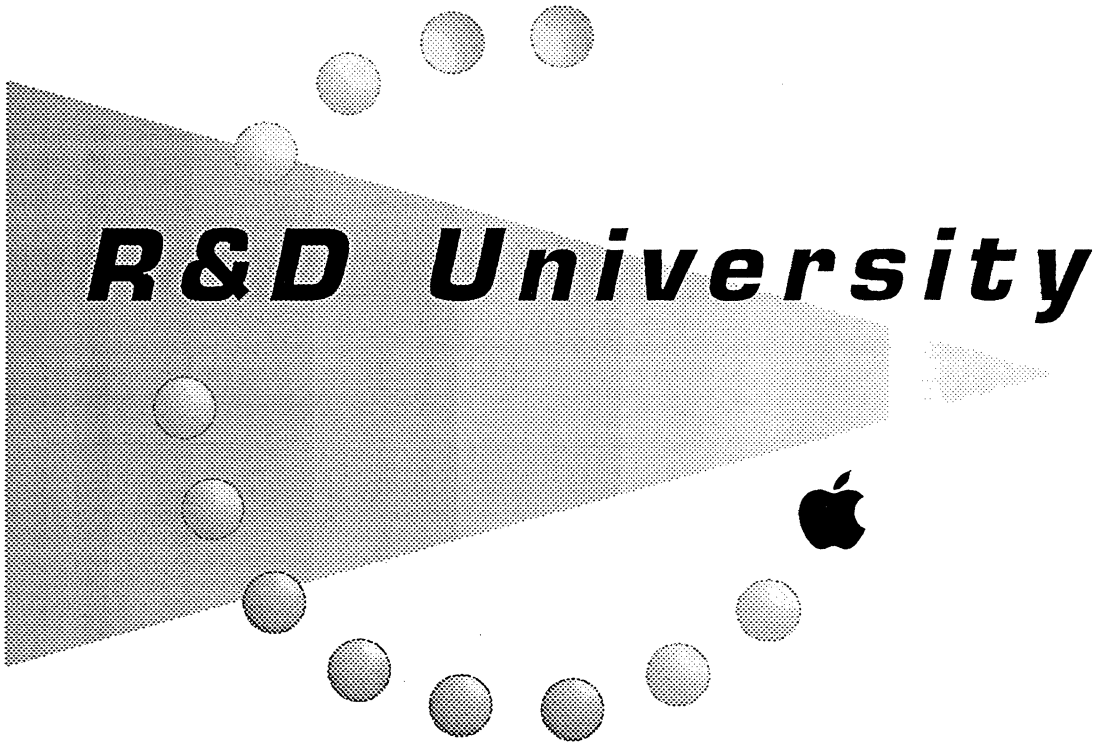
Introduction to PowerPC – Instruction Set



Instructor Ron Hocksprung

Intro to PowerPC video - Samuel

PowerPC Class - Architecture

PowerPC

PowerPC Architecture Overview

Book I - User Instruction Set
- Branch Unit
- Fixed Point Unit
- Floating Point Unit

Book II - Virtual Environment

Book III - Operating Environment

The First PowerPC Chips (Book IV)

First risc machine CDC-6600

PowerPC Class - Architecture

PowerPC

PowerPC Architecture Overview

Book I - User Instruction Set
- Branch Unit
- Fixed Point Unit
- Floating Point Unit

Book II - Virtual Environment

Book III - Operating Environment

The First PowerPC Chips (Book IV)

Ron Hochsprung 2/6/93

PowerPC Class - Architecture

RISC History and Lineages

IBM 801 Branch Processor

-> RT -> POWER -> PowerPC

Berkeley RISC Register Windows

-> 29K, SPARC

Stanford MIPS

-> MIPS, 88K, Alpha, ARM

RISC Fundamentals

PipeLined Programming Model
Compiler has Substantial Effect on Performance

Simple, Fixed Instruction Formats
1-Cycle Decode, Large Number of Registers

Simple Semantics
1-Cycle Execution Stage
Provide "Primitives" to Compiler

Only Loads & Stores Reference Memory
Load/Store Architecture

Minimize Use of Critical Resources
e.g., Condition Codes

Caches

- We'll look at several of these characteristics with real RISC examples.

Instruction Execution Model

F	1. Fetch Next Instruction
D	2. Decode Instruction, (Calculate Addresses, Fetch Operands, etc.)
X	3. Perform Operation
W	4. WriteBack Results
	5. Goto Step 1.

Ideal Instruction Sequence

Instructions Require only 1 Clock per Stage

1. Fetch Next Instruction	F	I1	-	-	-	I2
2. Decode Instruction, Calculate Addresses, Fetch Operands	D	-	I1	-	-	-
3. Perform Operation	X	-	-	I1	-	-
4. WriteBack Results	W	-	-	-	I1	-
5. Goto Step 1.						

Clocks

PipeLining

Increase Throughput by Keeping Resources Busy

Implement Each Instruction Stage by a Separate Unit - Pipe Stage.

Each Pipe Stage Operates on a Different Instruction in a Cycle

Don't Wait for An Instruction to Complete before Starting the Next

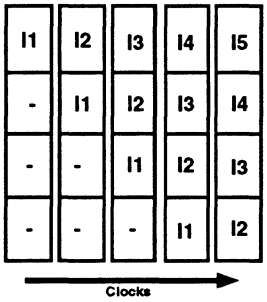
Does Not Reduce Latency!!

- Cycle Time of each Stage is the same.

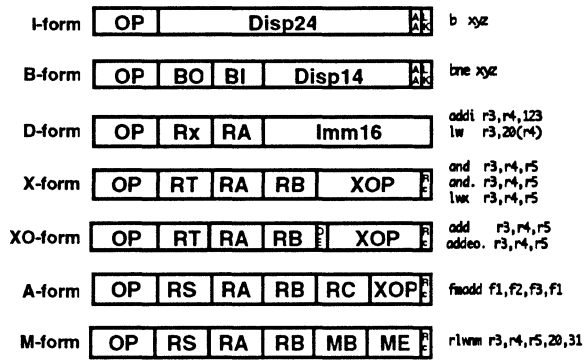
PipeLined Instruction Sequence

Instructions Require only 1 Clock per Stage

1. Fetch Next Instruction
2. Decode Instruction, Calculate Addresses, Fetch Operands
3. Perform Operation
4. WriteBack Results
5. Goto Step 1.

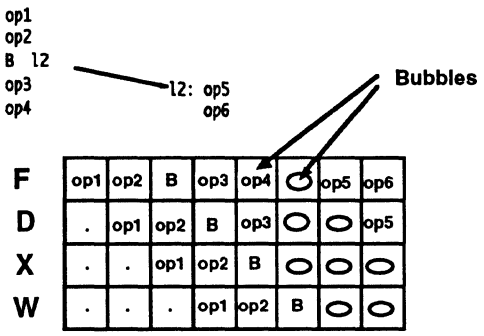


Instruction Formats



Branches - The Problem

Branches Leave "Bubbles" in PipeLine



SuperScalar RISC (Instructions/Clock > 1)

Super-Pipelining

PipeLine Stages Take Less Than One Clock

(R4000)

Multi-Issue

Issue more than One Instruction (to Multiple Units) per Clock

(88110,PowerPC,SuperSparc)

PowerPC - The Architecture

Derived from POWER

(Performance Optimized With Enhanced RISC)

Simplified; Removed Inhibitors to MultiScalar Implementations

Added Features Where Necessary: e.g. Synchronizing Ops, Single-Precision Ops

Extended for 64-bit Data & Address, Little-Endian Support

Defined by 4 "Books"

Book I - User Instruction Set Architecture

Book II - Virtual Environment

Book III - Operating Environment

Book IV - Implementation Features

PowerPC - Unusual Features

Branch Processor
Branch "Folding"

Mis-Aligned Loads/Stores
Supports 68K-Aligned Accesses

Load/Store Multiple
For Procedure Prolog/Epilog Code

Update Forms of Load/Store
Reduces Code in Loops

Move "Assist" Instructions
"String" Support

Synchronization Primitives
(LWARX / STWCX., EIEIO, SYNC)

Floating Multiply-Accumulate
Increased Floating Bandwidth

PowerPC

Architecture
vs
Implementation

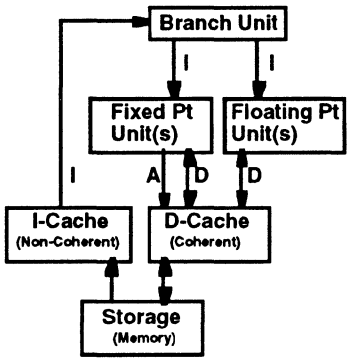
Some Architectural Features May Require Software Assist in An Implementation

String Ops
Load/Store Multiple
Floating Point "Hard" Cases (e.g., NaN)
TLB Reload (Page Table Walk)

The Code is Supplied by Implementors as Part of Book4

Possibly, by "Fast-Path" Interrupts with "Hidden" Hardware Support

PowerPC
Block Diagram



PowerPC

PowerPC Architecture Overview

Book I - User Instruction Set

- Branch Unit
- Fixed Point Unit
- Floating Point Unit

Book II - Virtual Environment

Book III - Operating Environment

The First PowerPC Chips (Book IV)

Book I - User Instruction Set

Defines Basic Programming Model for Compiler and/or Assembler Code

Instruction Set which MUST! be present to be called a "PowerPC" Chip

Can NOT affect any Privileged Resources

Does NOT discuss Caches or Virtual Memory
(I.e., a Simple Memory Model)

PowerPC

PowerPC Architecture Overview

Book I - User Instruction Set

- Branch Unit
- Fixed Point Unit
- Floating Point Unit

Book II - Virtual Environment

Book III - Operating Environment

The First PowerPC Chips (Book IV)

Branch Unit Concepts

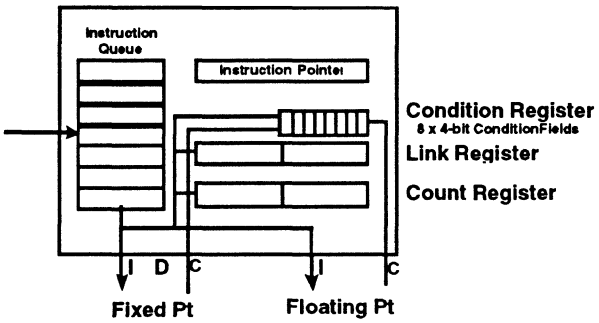
Branch Unit "Processes" Branches
Branches "Fold" Away

All The Information Necessary is Contained Within the Branch Unit
(Condition, Link, Count Registers)

Fixed / Floating Units "See" Only Fixed / Floating Instructions
Not Branches!

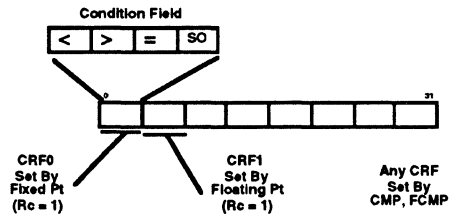
In Ideal Case, Branches are FREE!
Compiler Needs to "Schedule" Conditions and Branches

Branch Unit Diagram



Condition Register

8 Condition Fields
32 Bits for Conditional Branches



Should be Considered "Registers" for Scheduling of Branches

Link Register

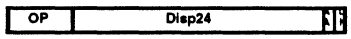
Set by Branch w/ LK=1
or, Move from Fixed Pt Reg (MTSPR)
Can by Copied to Fixed Pt Reg (MFSPR)
Typically used for Subroutine Linkage

Count Register

Set by Move from Fixed Pt Reg (MTSPR)
Altered by Branch w/ Decrement
Can by Copied to Fixed Pt Reg (MFSPR)
Typically used for Loop Counting, and Indirect Branching

Branch Instructions

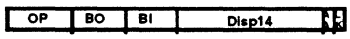
Unconditional, 24-bit Word Displacement

I-form 


```
b  xyz
bl subr
```

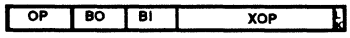
LK - Set Link Register to Next Instruction's Address
AA - Interpret Displacement as Absolute Address

Conditional, 14-bit Word Displacement

B-form 


```
bne xyz
blt 1,abc
```

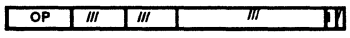
Conditional, thru LR or CTR

XL-form 


```
blr
beqctr
```

BO - Determines type of Branch (e.g., condition True/False)
BI - Condition Register Bit to be used as "condition"

System Call

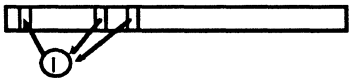
B-form 

Condition Register Logical Instructions

Operate upon Bits within the Condition Register

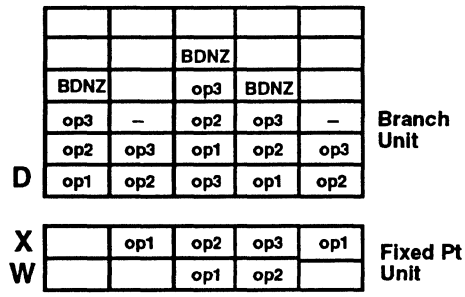
XL-form 

Example: CROR 1, 8, 11

CR 

Branch Folding

```
l1:
  op1
  op2
  op3
  BDNZ  l1
```

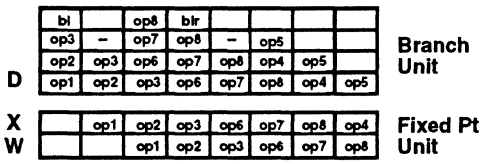


Branch Folding

Another Example

```
op1
op2
op3
bl  foo
op4
op5
```

```
foo:
  op6
  op7
  op8
  blr
```



The BL and BLR are Free!!

Branch Prediction

Used When Branch Unit Does Not Yet Know Direction of Branch

(e.g., Condition not Set Yet)

Uses Sign of Displacement to "Predict" if Branch Will Be Taken

((I<6> & I<8>) | I<16>) ^ I<10>

"Backwards" Branch Assumed Taken

(i.e., End-of-Loop Branches)

Software Can Invert Sense of Prediction

(e.g., Based on Tracing)

PowerPC

PowerPC Architecture Overview

Book I - User Instruction Set

- Branch Unit
- Fixed Point Unit
- Floating Point Unit

Book II - Virtual Environment

Book III - Operating Environment

The First PowerPC Chips (Book IV)

Fixed Point Unit Concepts

Contains 32 General Purpose Registers (GPRs)

Conceptually, Performs all Loads/Stores

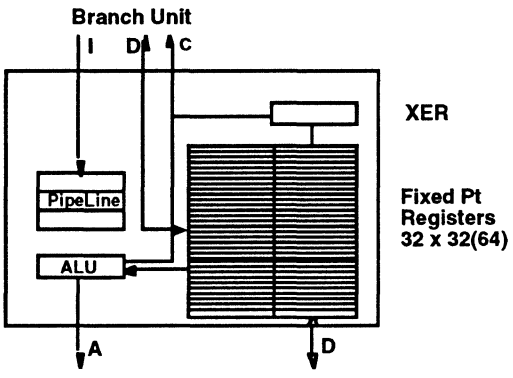
Executes all Fixed Point (Integer) Arithmetic

Executes all Logicals, Shifts and Rotates

Executes all Compares and Traps

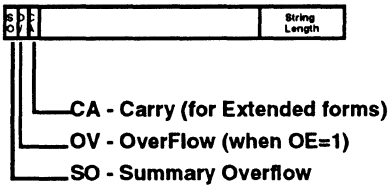
Special Instructions to Move Between GPRs and Branch Unit Registers

Fixed Point Unit

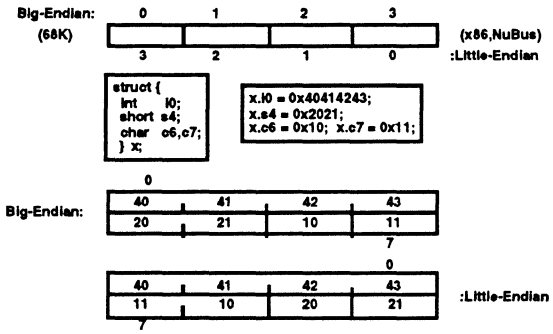


fiXed Exception Register (XER)

Contains Fixed Pt Related "Extra" Data

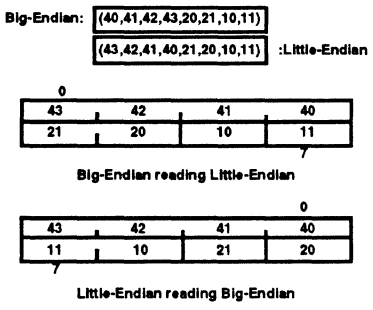


Endian-Ness



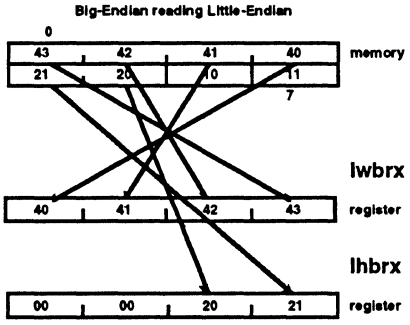
Relative Location of Elements is Different

Byte-Wise Ordering (e.g., to Disk)



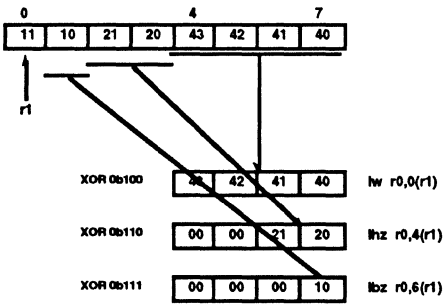
Individual Elements are "Byte Reversed"

Byte-Reverse Forms of Load/Store



Byte-Reversal "Swaps" Bytes on Element Size

Little-Endian Mode

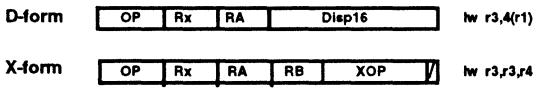


Little-Endian Mode "Swizzles" Address

Assumes that Memory has been Consistently Byte-Swapped (on Double-Word Basis)

Fixed Pt Loads & Stores

Base-Displacement & Indexed Forms
(with optional Update of Base)



Loads & Stores are Overlapped!!

(with Minimum of One Clock Latency)

Mis-Alignment is Auto-Magically Handled

(with possible performance penalty!)

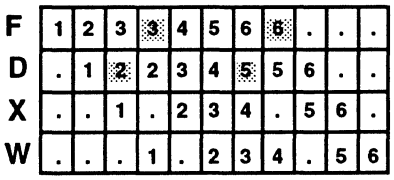
Byte-Reversed Forms for Mixed-Endian

Load Example

No Compiler Optimization

I += 123;
J -= 567;

```
1  lw   r3,I
2  addi r3,r3,#123
3  stw  r3,I
4  lw   r3,J
5  addi r3,r3,#-567
6  st   r3,J
```



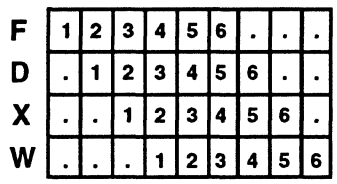
CPI = 1.3

Load Example

Compiler Optimization (Scheduling)

I += 123;
J -= 567;

```
1 lw   r3,I
2 lw   r4,J
3 addi  r3,r3,#123
4 addi  r4,r4,#-567
5 stw  r3,I
6 stw  r4,J
```



CPI = 1

Load & Store Multiple

LMW, STMW

Move from/to RS/RT through R31

Used for Subroutine Prolog/Epilog

Being "de-emphasized" in Architecture

Move Assist (Strings)

LSWI, LSWX, STSWI, STSWX

Load/Store Consecutive Bytes to/from Registers

Length is in Instruction or XER

Partial Loads are Zero-Filled

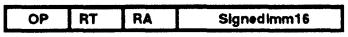
Synchronization Instructions

SYNC, EIEIO, LWARX, STWCX.

(Described in Book II Section)

Fixed Point Arithmetic

D-form



addi r3, r4, -1

XO-form

add r3, r4, r5

Differ on use of Carry
(I.e., ADDI vs ADDIC)

(use Carry forms only when necessary!!)

Can set CR Field 0 (Rc bit)

Can detect Overflow (OE-bit in XO-form)

(use Overfow Enable forms only when necessary!!)

Carry Usage Example

MultiPrecision Arithmetic

```
addc  r5,r5,r8     ; sets CA
adde  r4,r4,r7     ; uses, sets CA
```

Arithmetic Instructions Notes

Add Immediate (ADDI) Can't Add to R0!

Subtract From

Immediate form is Especially Useful

Multiply High

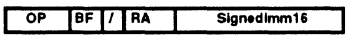
Fast Divide for Small Constants

```
addis  r4,0,0x5555      r4 = 1/3 (fraction)
addic  r4,r4,0x5556
mulhw  r3,r3,r4         r3 = r3/3
```


Compares

Sets any CR Field with Compare Results

D-form



cmpi 1, r4, -1

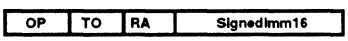
X-form

cmpi 3, r4, r5

Traps

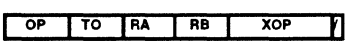
Compares, Traps if specified conditions (TO) are met

D-form



twnei r4, 0

X-form



twne r4, r5

PowerPC Class - Architecture

Logicals, Shifts, Rotates

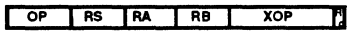
RA field specifies Result Reg!

Logicals

D-form

andi. r0, r1, 0x5

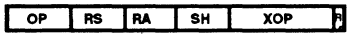
X-form



xor r3, r4, r5

Shifts

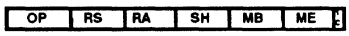
X-form



srai r0, r3, 16

Rotates

M-form



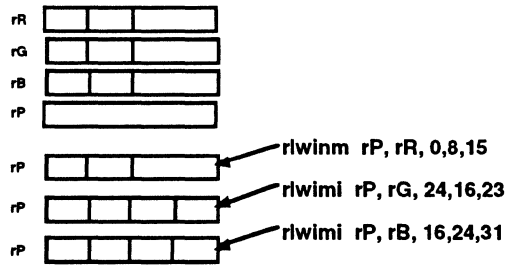
rlwinm r0, r3,0,27,31

Ron Hochsprung 2/6/93

PowerPC Class - Architecture

Rotate Example

This stuff is good for extracting bit fields



Ron Hochsprung 2/6/93

PowerPC Class - Architecture

"Special" Moves

Move To/From SPR - MTSPR/MFSPR

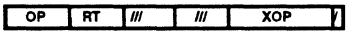
XFX-form

mfspr r0, LR

Move From Condition Reg - MFCR

mfcr

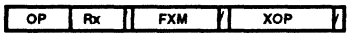
X-form



~~mfcr~~ r0

Move To CR Field - MTCRF

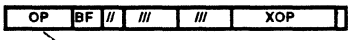
XFX-form



mtcrf 0x3, r3

Move To CR Field from XER - MCRXR

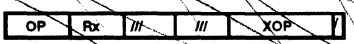
X-form



mcrxr 1

Move To/From PMR - MTPMR/MFPMR

X-form



mtpmr r0

Ron Hochsprung 2/6/93

PowerPC

PowerPC Architecture Overview

Book I - User Instruction Set

- Branch Unit
- Fixed Point Unit
- Floating Point Unit

Book II - Virtual Environment

Book III - Operating Environment

The First PowerPC Chips (Book IV)

Floating Point Unit Concepts

Contains 32 Floating Point Regs (FPRs) in Double-Precision (64-bit) Format

Sources & Sinks Floating Point Data for FP Stores & Loads

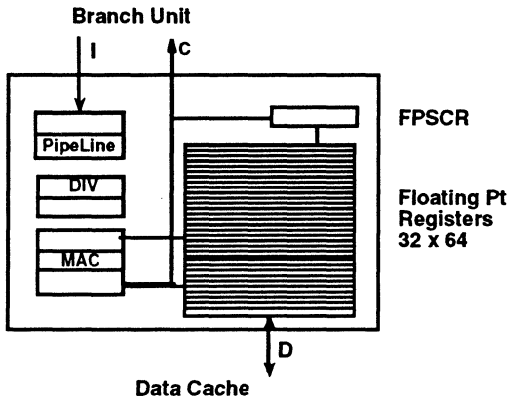
Single-Precision Operations Use SubSet of Double-Precision Data

Performs all Floating Point Arithmetic, Conversions & Compares

Multiply-Accumulate is Basic Arithmetic Element

No Direct Path Between Fixed Pt and Floating Pt

Floating Pt Unit



Floating Point Status and Control Register (FPSCR)

Contains Bits Which Control and Report Floating Point Operations

Rounding Mode

Exception Enables

Exceptions

etc.

Floating Point Instructions

Multiply-Add

A-form

fmadd f0, f1, f2,f3

FRT <- FRA * FRC + FRB

Misc Arithmetic

X-form

fctiw f0, f2

Note: Rc sets Summary Exception Bits from FPSCR

Floating Pt Compare

X-form

fcmpu 0, f1, f2

PowerPC

PowerPC Architecture Overview

Book I - User Instruction Set

- Branch Unit
- Fixed Point Unit
- Floating Point Unit

Book II - Virtual Environment

Book III - Operating Environment

The First PowerPC Chips (Book IV)

Book II - Virtual Environment

Introduces Additional Programming Model Concepts

General Concepts

Caches

Virtual Storage

Time Base (Real-Time Clock)

Multi-Processor Related

Atomicity

Globally Performed

Coherency

PowerPC Cache Concepts

Model Assumes Separate I & D Caches

Size and Granularity can be Different for I & D

Typical Cache Block Size is 32-64 Bytes

Coherency for Data Caches

Explicit Cache Management Instructions

Storage Attributes (WIMG Bits)

W - Write-Through

W = 1 -> All Stores MUST go to Memory
W = 0 -> Store-In is Allowed

I - Cache Inhibited

I = 1 -> All Loads/Stores MUST go to Memory
I = 0 -> Data may be Cached

M - Memory Coherency Required

M = 1 -> Data MUST be Maintained Consistent
M = 0 -> Coherency is NOT Required

G - Guarded Storage

G = 1 -> NO! Speculative access
G = 0 -> Speculative Loads allowed

Cache Instructions

I-Cache

Instruction Cache B lock Invalidate - ICBI

Instruction Sychronize - ISYNC

D-Cache

Data Cache Block Touch - DCBT

Data Cache Block Touch for Store - DCBTST

Data Cache Block set to Zero - DCBZ

Data Cache Block Store - DCBST

Data Cache Block Flush - DCBF

Atomicity of Storage Accesses

Only "Aligned", Scalar Accesses are Atomic

Move Assists, LWM/STWM, FP Doubles are NOT Atomic

Globally Performed

"Appears to be Complete"
With Respect to Other Processors & "Mechanisms"

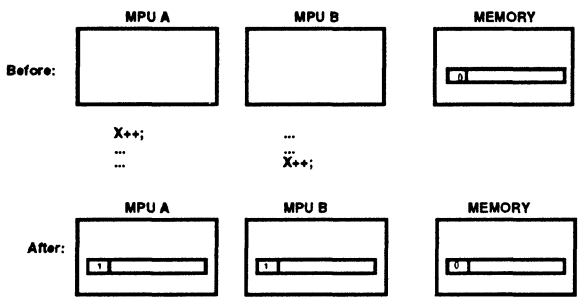
SYNC Instruction Guarantees Global Performance

Coherency

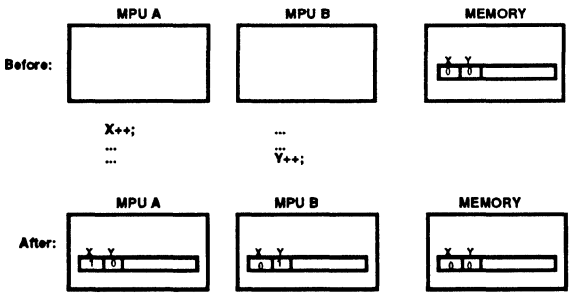
After a Coherent Storage Access is Globally Performed,
All Processors (Mechanisms) "See" Latest Version

Coherency applies to Cache Blocks

MP Caching, Without Coherency



MP Caching, Without Coherency



Coherency Mechanism

Coherency is done on Cache Blocks (e.g., 32 Bytes)

Cache Blocks have 4 possible States:

M - Modified
E - Exclusive
S - Shared
I - Invalid

Caches "Snoop" Bus Activity

Stores can only be done to Exclusive or Modifed Blocks

A change from Shared/Exclusive to Modified must be indicated on the Bus (so that it can be Snooped)

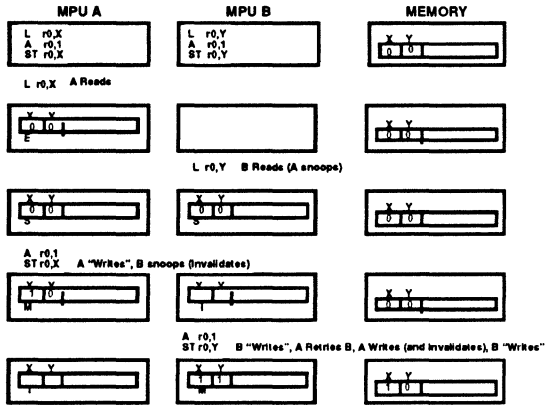
A Cache which Snoops a Read to an Exclusive Block changes the Block's State to Shared (and, informs the Reader).

A Cache which Snoops a Read of a Block which it has Modified, "Retries" the Reader, Writes its Modified Block to Memory and changes its Block's state to Invalid.

Subsequent re-Read will get latest copy from Memory

At most, One Cache has a Modified copy of a Block

MP Caching, with Coherency



Synchronization Primitives

Synchronize - SYNC

Guarantees that All Prior Loads & Stores Have Been Globally Performed.

i.e., can participate in Coherency Mechanism.

Enforce In-Order for I/O - EIEIO

(Order Storage Access - OSA)

Used to Separate Cache-Inhibited Loads & Stores to Ensure Program Order (on the Bus)

"Semaphore" Primitives

Load Word and Reserve (indeXed) - LWARX

Loads Word and Creates a "Reservation"
The Reservation is Associated with the Data Address of the LWARX

STore Word Conditional (indeXed) - STWCX.

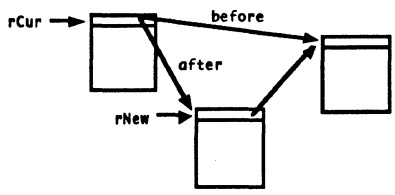
Iff a Reservation Exists, Performs the Store.
Sets CRF0 to Indicate Success; EQ=1 -> Store Made.
Unconditionally, Clears any Reservation.

A Reservation is "lost" When Coherency Detects Store to the Reservation's Address

```
Lock:
  lwarx  r4,0,r3   ; fetch current value
  addi   r0,0,-1   ; generate 1's
  cmpi   1,r4,0    ; check current == 0
  stwcx. r0,0,r3   ; store 1's (?)
  bne    Lock      ; lost reservation?
  beqlr  1         ; initial == 0
  b      Lock      ; try again
```


LWARX / STWCX. Example
Link List Update

```
Insert:
  lwarx  rT, NXT(rCur)
  stw    rT, NXT(rNew)
  sync
  stwcx. rNew, NXT(rCur)
  beqlr
  b      Insert
```



Time Base (PowerPC)

64-Bit Counter

Resolution is Implementation Dependent

User Instructions:

Move From Time Base - MFTB
Move From Time Base Upper - MFTBU

Real-Time Clock (601)

As Described in Book II 0.04

PowerPC

PowerPC Architecture Overview

Book I - User Instruction Set
- Branch Unit
- Fixed Point Unit
- Floating Point Unit

Book II - Virtual Environment

Book III - Operating Environment

The First PowerPC Chips (Book IV)

Book III -Operating Environment

Defines "Privileged" Instructions & Facilities

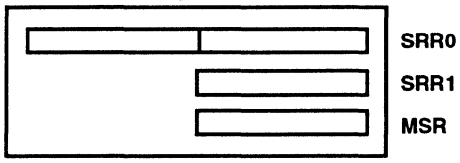
Storage Control (Virtual Memory)

Interrupts

Timing Facilities

Branch Unit Additions

Registers

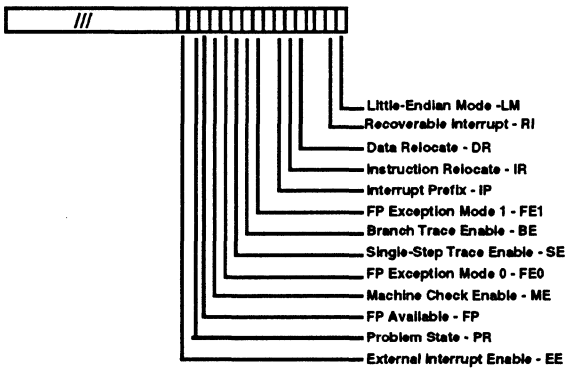


Save/Restore Register 0 - SRR0
Save/Restore Register 1 - SRR1
Machine State Register - MSR

Instruction

Return From Interrupt - RFI

Machine State Register - MSR



When an Interrupt is Taken

Save "Current" Instruction Address in SRR0

Copy MSR to SRR1 (possibly, setting SRR1<0:15>)

Modify MSR (EE = DR = IR = PR = 0)

Start Executing at Entry of Interrupt

Return From Interrupt (RFI)

Copy SRR1 to MSR

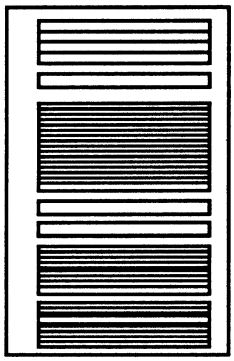
Start Execution at Address in SRR0

Interrupts are Precise

Except for Imprecise FP Exceptions

Fixed Point Unit Additions

Registers



Software-Use SPRs - SPRGn

Decrementer - DEC

Segment Registers (16x32)

Storage Description Register 1 - SDR1

Data Storage Interrupt Status Register - DSISR

Instruction Block Address Translation Registers (BATs)

Data Block Address Translation Registers (BATs)

Fixed Point Unit Additions

Instructions

Move To/From MSR - MTMSR, MFMSR

Data Cache Block Invalidate - DCBI

Move To/From Segment Register - MTSR/MFSR

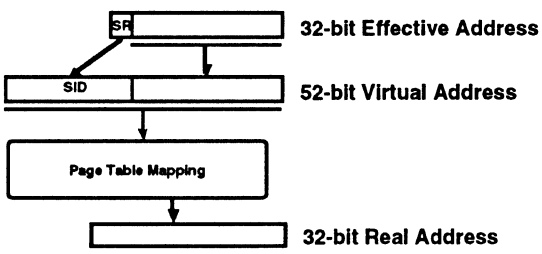
Move To/From Seg Reg Indirect - MTSRIN/MFSRIN

Translation Lookaside Buffer Entry Invalidate - TLBEI

(New SPRs for MTSPR/MFSPR)

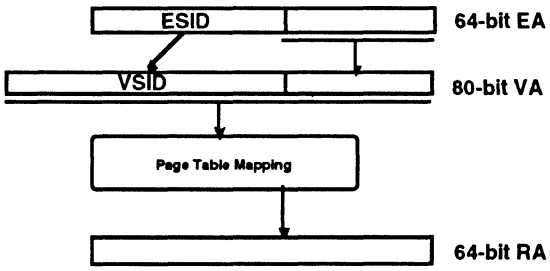
PowerPC

Storage Addressing Model (32-bit)



PowerPC

Storage Addressing Model (64-bit)



PowerPC

PowerPC Architecture Overview

Book I - User Instruction Set

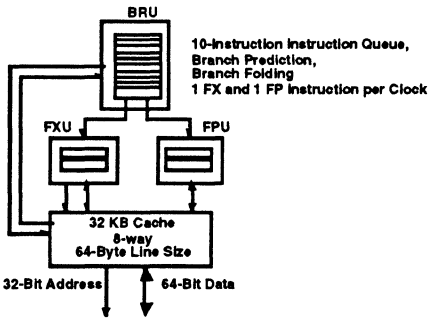
- Branch Unit
- Fixed Point Unit
- Floating Point Unit

Book II - Virtual Environment

Book III - Operating Environment

The First PowerPC Chips (Book IV)

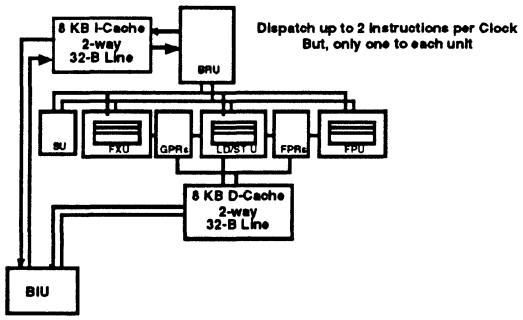
601 - The First (almost) PowerPC Chip



PowerPC Class - Architecture

603 - The Cheapest PowerPC Chip

Static Design - Low Power

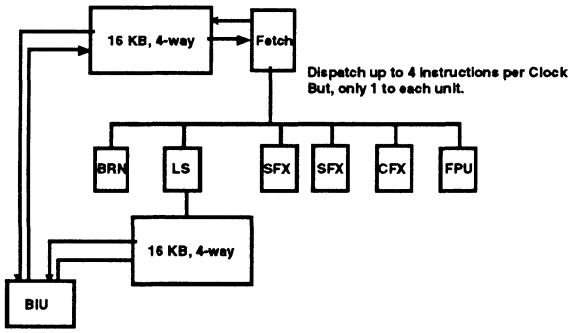


Dispatch up to 2 instructions per Clock
But, only one to each unit

Ron Hochsprung 2/6/93

PowerPC Class - Architecture

604 - The 601 Replacement

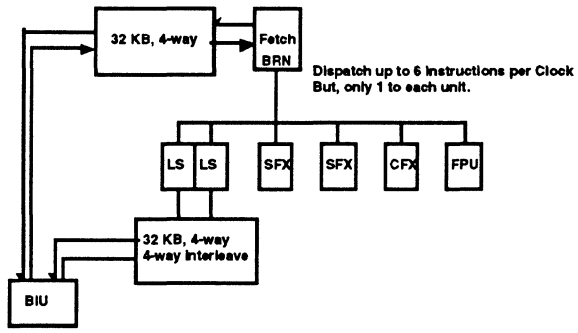


Dispatch up to 4 instructions per Clock
But, only 1 to each unit.

Ron Hochsprung 2/6/93

PowerPC Class - Architecture

620 - The First 64-Bit PowerPC Chip



Dispatch up to 6 instructions per Clock
But, only 1 to each unit.

Ron Hochsprung 2/6/93

PowerPC Programming Model

Table Of Contents (TOC)

Inter-Module Calls (Shared Libraries)

Comments on Optimization

Code Examples

Register Conventions

Branch Unit Regs

```
LR  - volatile
CTR - volatile
CR  - Fields 2-5 non-volatile
```

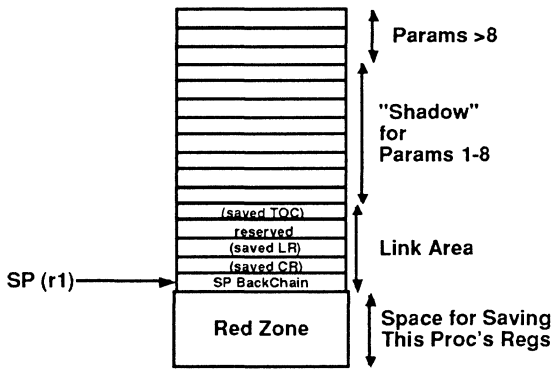
Fixed Pt Regs

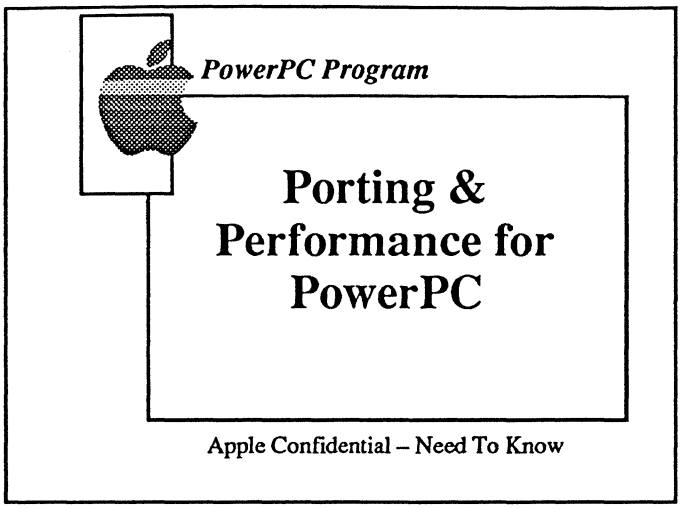
```
0     - scratch/epilog/prolog
1     - Stack Pointer
2     - Table Of Contents (TOC) ptr
3:10  - Argument/scratch
11    - scratch/function ptr
12    - scratch/epilog/prolog
13:31 - locals (non-volatile)
```

Floating Pt Regs

```
0     - scratch/epilog/prolog
1:13  - Argument/scratch
14:31 - locals (non-volatile)
```


Stack Frame

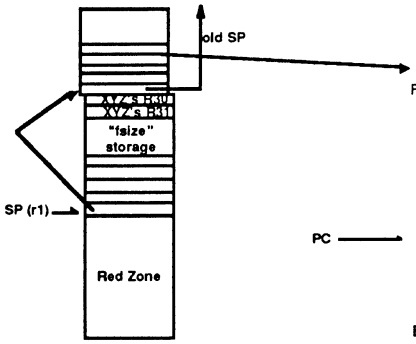




Porting & Performance for PowerPC

Proc Call

Back From BAR

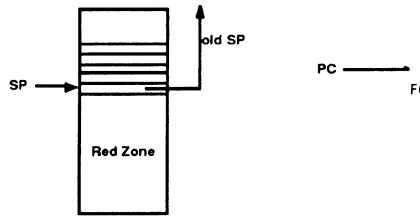



```
XYZ:
  lw  r3,arg1
  lw  r4,arg2
  bl  FOO
  ...
FOO:
  mflr  r0
  stmw  r30,-8(r1)
  stw   r0,8(r1)
  stwu  r1,-fsize(r1)
  ....
  bl    BAR
  ....
PC ->
  lw    r0,fsize+8(r1)
  lw    r1,0(r1)
  mtlr  r0
  lmw   r30,-8(r1)
  blr
BAR:
  ...
  blr
```


PowerPC - Software Notes

Proc Call

Back From FOO




```
XYZ:
  lw  r3,arg1
  lw  r4,arg2
  bl  FOO
PC ->
  ...
FOO:
  mflr  r0
  stmw  r30,-8(r1)
  stw   r0,8(r1)
  stwu  r1,-fsize(r1)
  ....
  bl    BAR
  ....
  lw    r0,fsize+8(r1)
  lw    r1,0(r1)
  mtlr  r0
  lmw   r30,-8(r1)
  blr
BAR:
  ...
  blr
```


PowerPC - Software Notes

Table Of Contents (TOC)

Each Module Has Its Own
in Data Area (RW)

Contains:
Module's Statics
Procedure Descriptors

Created at Link Time
Filled In as Part of Program Loading

Saved/Restored Across
Inter-Module Calls
Used by Shared Library Mechanism

Procedure Descriptors

Used for "Pointer to Function"
and Inter-Module Calls

The Pointer is Address of Descriptor
(in Target's TOC)

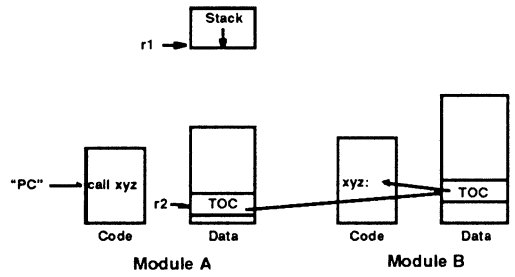
Descriptor:

Code Address
TOC address
Environment Pointer

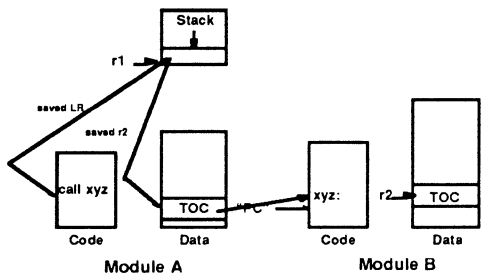
```
lw  r11,foo.descr(rTOC)
bl  $PTRGL
lw  rTOC, 20(rSP)
```

```
$PTRGL:
  lw   r0, 0(r11)
  stw  rTOC, 20(rSP)
  mtctr r0
  lw   rTOC, 4(r11)
  bctr
```


Intermodule Call
Before



Intermodule Call
After



Comments on Code Optimization

"Well Known" Optimizations Are Applicable,
And, Have More Affect
Than Instruction Scheduling

Global Optimization
Common Sub-Expressions
Strength Reduction

General Optimizations Which are
Especially Useful for RISC

Loop UnRolling
Register Allocation
Alignment Considerations

Optimization must be Tempered with
Code/Data Expansion "Hit"

PowerPC Specific Optimizations

Minimize Loads
(i.e., Keep Data in Registers)

Minimize Branches
(i.e., Large Basic Blocks, inLining)

Schedule Loads as Early as Possible from Their Use

Schedule Condition Setting as Early as Possible from Its Use
(i.e., Conditions are like Branch Unit "Loads")

"Shuffle" Independent Code Sequences
(i.e., Inter-Schedule Dependencies)

Use MULT by Reciprocal in Place of DIV
(e.g., MULH on Fixed Pt)

Shading Example (Inner Loop Only)

```
loop:
  rlwinm rP, rR, 0,8,15      ; Red
  add    rR, rR, rRd         ; Next Red
  rlwimi rP, rG, 24,16,23    ; Green
  add    rG, rG, rGd         ; next Green
  rlwimi rP, rB, 16,24,31    ; Blue
  add    rB, rB, rBd         ; next Blue
  stwu   rP, 4(rPptr)        ; stash it
  bdn    loop                ; around the loop
```

601/603: 7 clocks

604: 3 clocks

620: 2 clocks

Compiler Listing - div3

int div3(int i) { return i/3; }

```
GPR's set/used:  s--s ---- ---- ----   ---- ---- ---- ----
FPR's set/used:  ---- ---- ---- ----   ---- ---- ---- ----
 CR's set/used:  ---- ----

    | 000000                          PDEF    div3
    | 000000                          PROC    i,r3
   2| 000000 cau    3C00 5555   1     LIU     r0=21845
   2| 000004 ai     3800 5556   1     AI      r0=r0,21846
   2| 000008 mul    7C00 18D6   5     MUL     r0=r0,r3,mq"
   2| 00000C rlinm  5403 0FFE   1     SRL     r3=r0,31
   2| 000010 a      7C60 1814   1     A       r3=r0,r3
   1| 000014 bcr    4E80 0020   0     BA      lr
      Straight-line exec time    9
```


Compiler Listing - bsf.c

```
double bsf( double *dp, double *cp ) {
  int i; double X = 0.0;
  for( i=0; i<64; i++ )
    X += *dp++ * *cp++;
  return( X );
  }

    | 000000                          PDEF    bsf
    | 000000                          PROC    dp,cp,r3,r4
   4| 000000 l      80A2 0000   1     L       r5=.+bsf(r2,0)
   0| 000004 ai     3084 FFF8   1     AI      r4=r4,-8
   4| 000008 lfd    C825 0000   1     LFL     fp1=+bsf(r5,0)
   0| 00000C cal    38A0 0040   1     LI      r5=64
   0| 000010 mtspr  7CA9 03A6   1     LCTR    r5
   0| 000014 ai     3063 FFF8   1     AI      r3=r3,-8
                                  CL.0:
   6| 000018 lfdu   CC03 0008   1     LFDU    fp0,r3=(double)(r3,8)
   6| 00001C lfdu   CC44 0008   1     LFDU    fp2,r4=(double)(r4,8)
   6| 000020 fma    FC20 088A   1     FMA     fp1=fp1,fp0,fp2
   5| 000024 bc     4200 FFF4   0     BCT     CL.0
                                  CL.3:
   1| 000028 bcr    4E80 0020   0     BA      lr
```

PowerPC User Instruction Set Architecture

Book I

Version 1.02

January 8, 1993

Distribution for IBM: softcopy on KISS64

Owner: Jack Kemp
KEMP at AUSVM6
E64S/4A-015
IBM Corporation
Austin, TX 78758
Tele 512-838-1846
Tie Line 678-1846

Technical Content: Ed Silha
silha@austin.ibm.com
E22S/4F-019
IBM Corporation
Austin, TX 78758
Tele 512-838-1848
Tie Line 678-1848

IBM Confidential

NOTES:

- This is a controlled document.
- Verify version and completeness prior to use.
- See the Preface for additional important information.

Preface

This document defines the PowerPC User Instruction Set Architecture. It covers the base instruction set and related facilities available to the application programmer.

Other related documents define the PowerPC Virtual Environment Architecture, the PowerPC Operating Environment Architecture, and PowerPC Implementation Features. The PowerPC Virtual Environment Architecture defines the storage model and related instructions and facilities available to the application programmer, and the Time Base as seen by the application programmer. The PowerPC Operating Environment Architecture defines the system (privileged) instructions and related facilities. A PowerPC Implementation Features document defines the implementation-dependent aspects of a particular implementation.

The PowerPC Architecture consists of the instructions and facilities described in the PowerPC User Instruction Set Architecture, PowerPC Virtual Environment Architecture, and PowerPC Operating Environment Architecture documents. However, the complete description of the PowerPC Architecture as instantiated in a given implementation includes also the material in the PowerPC Implementation Features document for that implementation.

User Responsibilities

- Do not make any unauthorized alterations to the document (user notes permitted).
- Verify the version prior to use. Version verification procedure is described below.
- Verify completeness prior to use. The last page is labeled 'Last Page - End of Document'. The end of the Table of Contents shows the last page number. All pages are numbered sequentially.
- Report any deviations from these procedures to the document owner.

Next Scheduled Review

The next review is expected to be approximately in March, 1993. At least four weeks before this meeting, a DRAFT version of this document will be distributed.

Version Verification for IBM

- Link to the KISS64 disk in Yorktown or a shadow of this disk. In Yorktown, linking to KISS64 can be done with the command "GIME KISS64."
- Browse the newest file with a name of the form "PPC2xxxx LIST3820," by using the "browse" command.
- Verify that your version matches this file.

If your version is not current, please contact the document owner.

Version Verification for Other Firms

To be supplied.

Approval Process

The following procedure is followed for all changes to the content of this document:

- The Power Open Architecture Work Group (PAWG) meets quarterly or more frequently if necessary.
- At least four weeks before a meeting, a version of this document is distributed to the PAWG. It is marked DRAFT. Proposed changes are included and identified with change bars.
- The PAWG meets and decides each issue.
- Final alterations to this document are made, change bars are removed, and the entire document is distributed with a new version number and the word DRAFT removed.
- At the meeting or a subsequent one, new issues are discussed.
- The resulting changes are described in a new version of this document which is derived from the last non-DRAFT version. Proposed changes are identified with change bars, and the document is distributed to the PAWG. This document has a new version number and is marked DRAFT.
- The cycle repeats from the beginning.

Approvals

This version has been approved for user review by the document owner.

Table of Contents

Figures

Incomplete as of 1993/01/08

topic	reason	page
Make documents easy to read by people who are interested in 32-bit only machines.	Agreed at several PowerPC meetings.	
Jan Stone's complex programming examples should be added to Appendix E.1, Synchronization. Additional programming examples should be added to Appendix E.3, Floating-Point Conversions.		153, 159

Changes as of 1993/01/08 Version 1.02

change	reason	page
Delete sentence "In 32-bit mode, the high-order 32 bits of the next instruction address are set to 0" (four places).	Redundant and possibly confusing.	12+1
Delete RTL that shows clearing of the high-order 32 bits of the NIA and LR for 64-bit implementations in 32-bit mode.	Redundant and possibly confusing.	20, 21
Change ***mull*** to ***mullw*** (*Multiply Low Word*), and add ***mulld*** (*Multiply Low Doubleword*). Proposal put in early.	Was difficult to compute OV for ***mull*** when in 32-bit mode.	55
Change ***xor*** to ***nor*** in example.	Typo.	66
For disabled overflow exception, change "$FPSCR_{FR\ FI}$ are set to zero" to "$FPSCR_{FR}$ is set to one if the result is incremented when rounded, and otherwise to zero" and "$FPSCR_{FI}$ is set to one."	Correction.	95
Change Rios-2 mnemonics from fcvir/fcvirz to fcir/fcirz.	Tracking Rios-2 change.	113, 171
Remove the explicit grouping of the optional instructions, and add a remark that there are certain defined groups.	Agreed at Dec. 2 Power Open meeting.	119
Change Architecture Note for ***stfiwx*** to say that it may eventually be a required instruction.	Agreed at Dec. 2 Power Open meeting.	120
Change disabled exponent overflow case of ***frsp*** model regarding how FPSCR bits FR and FI are set.	Correction.	125
Make floating-point convert to integer model show that VXSNAN is set if the operand is an SNaN.	Agreed at Dec. 2 Power Open meeting.	128+1
Delete references to ***lock, lockd, lockrel.***	These have been deleted from Rios-2.	172, 186+1
Add section "Instructions with Different Semantics" (***dcbz*** and ***tblie***).	Agreed at Dec. 2 Power Open meeting.	173+1

Changes as of 1992/10/28

change	reason	page
Add Little-Endian mode, done via a hack on the low three bits of the EA rather than by actually reversing bytes. See Appendix D, "Little-Endian Byte Ordering" on page 145 for details. References to this appendix placed at start of sections containing load and store instructions.	Austin meeting, 20 October 1992.	145ff, and others

Changes as of 1992/10/05 Version 1.01 DRAFT

change	reason	page
Clarify that ***sync*** need not discard prefetched instructions.	This has confused people (***sync*** is not context synchronizing).	48
Add item to list of general synchronization notes in the Programming Examples appendix, warning against looping on a ***lwarx*** that fails to return a desired value.	Suggested by Mike Yamamura at 8 Aug. 1992 meeting at Apple. In some implementations such looping may flood the bus.	155
Add a caveat to the discussion of instruction completion in the "Instruction Fetching" section, citing the "Synchronization Requirements for Special Registers" appendix in Book III.	Truth.	15

Changes as of 1992/09/29

change	reason	page
Show that FR and FI are set to 0 by ***frsp*** of infinities and QNaNs, in the "Floating-Point Model" appendix.	Requested by Barry Dorfman. These were the only cases in the appendix for which FR and FI settings were not specified. Setting them to 0 is consistent with the definition of these bits.	123
Make floating-point terminology consistent as follows. ▪ Use "single/double format" for operand *formats*. ▪ Use "single/double-precision" for operand *values*. These changes are not marked with change bars.	Previously we sometimes said "single/double-precision format." IEEE uses "single/double format," and these terms nicely suggest the amount of storage the formats require. IEEE also uses "double operand," etc., but that sounds funny (does a double operand have two instances?).	various

Changes as of 1992/09/28

change	reason	page
Make symbols cr0, cr1, ..., cr7 (used with extended mnemonics) always have values 0, 1, ..., 7.	Fix inconsistency, pointed out by Ron Hochsprung and Mike Corrigan, in which the symbols sometimes had values 0, 4, ..., 28.	134
Change basic mnemonic generated by the ***mr*** extended mnemonic from ***ori*** to ***or***. Show it as example with the ***or*** instruction.	Suggested by Ron Hochsprung. Permits a "recording" variant.	144, 65
Add extended mnemonics for sundry CR Logical operations and for complementing a GPR. Show these as examples with the corresponding basic instructions (***cror***, ***crxor***, ***crnor***, ***creqv***, and ***nor***).	Suggested by Ron Hochsprung.	137, 144
Add examples, in the Extended Mnemonics appendix, for the *Rotate and Shift* and *Move To/From Special Purpose Register* extended mnemonics.	Omission was oversight.	141ff
Clarify that the SO bit of the XER is cleared only when software executes ***mtspr*** (to the XER) or ***mcrxr***.	Suggested by Andy Wottreng. Previous wording confused some people.	28
State that early implementations must implement XER bits 16:23 and allow these bits to be read and written by software in the normal manner.	Needed for compatibility with Power, as pointed out by Ron Hochsprung.	28, 167
Note incompatibilty with Power with respect to use of MSR bit 20 to control floating-point interrupts.	Omission was oversight.	169
Explain the seeming discrepancy between the extended opcode shown for ***sradi*** in the instruction description (413) and that shown in the opcode maps (826 and 827).	Some people were confused.	179
Eliminate from instruction descriptions unnecessary statements of the form "x is unchanged." (Some of these changes are not flagged, because they occur inside macros which may themselves occur within flagged areas.)	Such statements did not appear consistently. Moreover, they sometimes caused confusion (e.g., for ***mtfsfi*** the statement falsely implied that the FPSCR summary bits were not affected).	various
Clarify what it means for a NaN to be representable in single format.	Omission was oversight.	89
Clarify which floating-point operations cause an exception when an operand is an SNaN.	Requested by Andy Wottreng.	93
Show Rios-2 mnemonics for ***fctiw*** and ***fctiwz***.	Requested by Mark Rogers.	113
Note incompatibilities with Rios-2 for ***fctiw*** and ***fctiwz***.	Omission was oversight.	171

Changes as of 1992/09/23

change	reason	page
Add section to Programming Examples appendix showing uses of ***fsel***.	It's tricky to use if NaNs, infinities, or IEEE compatibility are important.	162
Revise discussion of Real-Time Clock incompatibilities with Power, to reflect the changes to the Time Base instructions.	Decided at 9-11 Sept. 1992 PowerPC architecture meeting.	165

Changes as of 1992/09/22

change	reason	page
Specify that the high-order 32 bits of instruction addresses are always 0 in 32-bit mode.	Decided at 9-11 Sept. 1992 PowerPC architecture meeting.	12
Note as Power incompatibility the fact that ***isync*** is now stronger than in Power (***ics***).	Decided at 9-11 Sept. 1992 PowerPC architecture meeting.	165

Changes as of 1992/09/21

change	reason	page
Eliminate ***lmd*** and ***stmd***.	Decided at 9-11 Sept. 1992 PowerPC architecture meeting.	42
Add a subsection to Section 1.9, Forms of Defined Instructions, describing the handling of optional instructions that are not implemented. Add a bullet to Section 1.10, Exceptions, doing same.	Decided at 9-11 Sept. 1992 PowerPC architecture meeting.	11

Changes as of 1992/09/18

change	reason	page
For ***sync***, cite Book III's discussion of TLB invalidates.	Decided at 9-11 Sept. 1992 PowerPC architecture meeting.	48
Make ***fres*** and ***frsqrte*** set FPSCR bits FR and FI to undefined values, rather than preserve them.	Decided at 9-11 Sept. 1992 PowerPC architecture meeting. Ease of implementation.	121
Make the VXSQRT bit of the FPSCR defined even if the implementation does not support either of the instructions that can set it (***fsqrt*[s] and ***frsqrte***).	Decided at 9-11 Sept. 1992 PowerPC architecture meeting. Provides uniform interface to software for reflecting and handling square root exceptions.	85
Move the discussion of Power compatibility for ***lmw*** and ***stmw*** to the "Incompatibilities with the Power Architecture" appendix, and cite that appendix in the *Load/Store Multiple* chapter. Correct the discussion to permit the implementation to execute an unaligned ***lmw*** or ***stmw*** correctly, without causing Alignment interrupt.	That's where Power compatibility considerations belong.	42, 165
Add RTCU and RTCL to the cases for which ***mfspr*** must give an Illegal Instruction type Program interrupt in early implementations for Power compatibility.	Omission was oversight.	165

Changes as of 1992/09/17

change	reason	page
Move the ***stfiwx*** instruction to Appendix A, "Optional Instructions" on page 119, and make it optional.	Decided at 9-11 Sept. 1992 PowerPC architecture meeting.	120
Move the ***fsel*** instruction to Appendix A, "Optional Instructions" on page 119, and make it optional. Revise the definition so that it selects based on a comparison with 0.0, instead of on a sign bit.	Decided at 9-11 Sept. 1992 PowerPC architecture meeting.	122

Changes as of 1992/09/16

change	reason	page
Eliminate PMR.	Decided at 9-11 Sept. 1992 PowerPC architecture meeting.	various

Chapter 1. Introduction

1.1 Overview

This chapter describes computation modes, compatibility with the Power Architecture, document conventions, a processor overview, instruction formats, storage addressing, and instruction fetching.

1.2 Computation Modes

The PowerPC Architecture allows for the following types of implementation:

- 64-bit implementations, in which all registers except some Special Purpose Registers are 64 bits long, and effective addresses are 64 bits long. All 64-bit implementations have two modes of operation: 64-bit mode and 32-bit mode. The mode controls how the effective address is interpreted, how status bits are set, and how the Count Register is tested by *Branch Conditional* instructions. All instructions provided for 64-bit implementations are available in both modes.
- 32-bit implementations, in which all registers except Floating-Point Registers are 32 bits long, and effective addresses are 32 bits long.

Instructions defined in this document are provided in both 64-bit implementations and 32-bit implementations unless otherwise stated. Instructions that are provided only for 64-bit implementations are illegal in 32-bit implementations, and vice versa.

1.2.1 64-bit Implementations

In both 64-bit mode and 32-bit mode of a 64-bit implementation, instructions that set a 64-bit register affect all 64 bits, and the value placed into the register is independent of mode. In both modes, effective address computations use all 64 bits of the relevant registers (General Purpose Registers, Link Register, Count Register, etc.), and produce a 64-bit result. However, in 32-bit mode, the high-order 32 bits of the computed effective address are ignored when

accessing data, and are set to 0 when fetching instructions.

1.2.2 32-bit Implementations

For a 32-bit implementation, all references to 64-bit mode in this document should be disregarded. The semantics of instructions are as shown in this document for 32-bit mode in a 64-bit implementation, except that in a 32-bit implementation all registers except Floating-Point Registers are 32 bits long. Bit numbers for registers are shown in braces ({ }) when they differ from the corresponding numbers for a 64-bit implementation, as described in Section 1.5.1, "Definitions and Notation" on page 2.

1.3 Instruction Mnemonics and Operands

The description of each instruction includes the mnemonic and a formatted list of operands. Some examples are the following.

```
stw     RS,D(RA)
addis   RT,RA,SI
```

PowerPC-compliant assemblers will support the mnemonics and operand lists exactly as shown. They will also provide certain extended mnemonics, as described in Appendix C, "Assembler Extended Mnemonics" on page 133.

1.4 Compatibility with the Power Architecture

The PowerPC Architecture provides binary compatibility for Power application programs, except as described in Appendix G, "Incompatibilities with the Power Architecture" on page 165.

Many of the PowerPC instructions are identical to Power instructions. For some of these the PowerPC instruction name and/or mnemonic differs from that in Power. To assist readers familiar with the Power Architecture, Power mnemonics are shown with the individual instruction descriptions when they differ from the PowerPC mnemonics. Also, Appendix F, "Cross-Reference for Changed Power Mnemonics" on page 163, provides a cross-reference from Power mnemonics to PowerPC mnemonics for the instructions in this document.

1.5 Document Conventions

1.5.1 Definitions and Notation

The following definitions and notation are used throughout the PowerPC Architecture documents.

- A program is a sequence of related instructions.
- Quadwords are 128 bits, doublewords are 64 bits, words are 32 bits, halfwords are 16 bits, and bytes are 8 bits.
- All numbers are decimal unless specified in some special way.
  - 0bnnnn means a number expressed in binary format.
  - 0xnnnn means a number expressed in hexadecimal format.

  Underscores may be used between digits.
- RT, RA, R1, ... refer to General Purpose Registers.
- FRT, FRA, FR1, ... refer to Floating-Point Registers.
- (x) means the contents of register x, where x is the name of an instruction field. For example, (RA) means the contents of register RA, and (FRA) means the contents of register FRA, where RA and FRA are instruction fields. Names such as LR and CTR denote registers, not fields, so parentheses are not used with them. Also, when register x is assigned to, parentheses are omitted.
- (RA|0) means the contents of register RA if the RA field has the value 1-31, or the value 0 if the RA field is 0.
- Bits in registers, instructions, and fields are specified as follows.
  - Bits are numbered left to right, starting with bit 0.
  - Ranges of bits are specified by two numbers separated by a colon (:). The range p:q consists of bits p through q.
  - For registers that are 64 bits long in 64-bit implementations and 32 bits long in 32-bit implementations, bit numbers and ranges are specified with the values for 32-bit implementations enclosed in braces ({ }). {} means a bit that does not exist in 32-bit implementations. {:} means a range that does not exist in 32-bit implementations.
- X_p means bit p of register/field X.
  $X_{p(r)}$ means bit p of register/field X in a 64-bit implementation, and bit r of register/field X in a 32-bit implementation.
- $X_{p:q}$ means bits p through q of register/field X.

$X_{p:q(r:s)}$ means bits p through q of register/field X in a 64-bit implementation, and bits r through s of register/field X in a 32-bit implementation.

- $X_{p\ q\ ...}$ means bits p, q, ... of register/field X. $X_{p\ q\ ...\ (r\ s\ ...)}$ means bits p, q, ... of register/field X in a 64-bit implementation, and bits r, s, ... of register/field X in a 32-bit implementation.
- ¬(RA) means the one's complement of the contents of register RA.
- Field i refers to bits 4xi to 4xi+3 of a register.
- A period (.) as the last character of an instruction mnemonic means that the instruction records status information in certain fields of certain Special Purpose Registers as a side effect of execution, as described in Chapter 2 through Chapter 4.
- The symbol || is used to describe the concatenation of two values. For example, 010 || 111 is the same as 010111.
- x^n means x raised to the n^{th} power.
- $^n x$ means the replication of x, n times (i.e., x concatenated to itself n−1 times). $^n 0$ and $^n 1$ are special cases:
  - $^n 0$ means a field of n bits with each bit equal to 0. Thus $^5 0$ is equivalent to 0b00000.
  - $^n 1$ means a field of n bits with each bit equal to 1. Thus $^5 1$ is equivalent to 0b11111.
- Positive means greater than zero.
- Negative means less than zero.
- A system library program is a component of the system software that can be called by an application program using a *Branch* instruction.
- A system service program is a component of the system software that can be called by an application program using a *System Call* instruction.
- The system trap handler is a component of the system software that receives control when the conditions specified in a *Trap* instruction are satisfied.
- The system error handler is a component of the system software that receives control when an error occurs. The system error handler includes a component for each of the various kinds of error. These error-specific components are referred to as the system alignment error handler, the system data storage error handler, etc.
- Each bit and field in instructions, and in status and control registers (XER and FPSCR) and Special Purpose Registers, is either defined or reserved.
- /, //, ///, ... denotes a reserved field in an instruction.
- Latency refers to the interval from the time an instruction begins execution until it produces a result that is available for use by a subsequent instruction.
- Unavailable refers to data or instruction storage that an instruction cannot access for any reason.

1.5.2 Reserved Fields

All reserved fields in instructions should be zero. If they are not, the instruction form is invalid: see Section 1.9.2, "Invalid Instruction Forms" on page 11.

The handling of reserved bits in status and control registers (XER and FPSCR) and in Special Purpose Registers (and Segment Registers: see Book III, *PowerPC Operating Environment Architecture*) is implementation dependent. For each such reserved bit, an implementation shall either:

- ignore the source value for the bit on write, and return zero for it on read; or
- set the bit from the source value on write, and return the value last set for it on read.

Programming Note

It is the responsibility of software to preserve bits that are now reserved in status and control registers and in Special Purpose Registers (and Segment Registers: see Book III, *PowerPC Operating Environment Architecture*), as they may be assigned a meaning in some future version of the architecture or in Book IV, *PowerPC Implementation Features* for some implementation. In order to accomplish this preservation in implementation independent fashion, software should do the following.

- Initialize each such register supplying zeros for all reserved bits.
- Alter (defined) bit(s) in the register by reading the register, altering only the desired bit(s), and then writing the new value back to the register.

When a currently reserved bit is subsequently assigned a meaning, every effort will be made to have the value to which the system initializes the bit correspond to the "old behavior."

1.5.3 Description of Instruction Operation

A formal description is given of the operation of each instruction. In addition, the operation of most instructions is described by a semiformal language at the register transfer level (RTL). This RTL uses the notation given below, in addition to the definitions and notation described in Section 1.5.1, "Definitions and

Notation" on page 2. RTL notation not summarized here should be self-explanatory.

The RTL descriptions do not imply any particular implementation.

The RTL descriptions do not cover the following:

- "Standard" setting of the Condition Register, Fixed-Point Exception Register, and Floating-Point Status and Control Register. "Non-standard" setting of these registers (e.g., the setting of Condition Register Field 0 by the ***stwcx.*** instruction) is shown.
- Invalid instruction forms.

Notation	Meaning
←	Assignment
¬	NOT logical operator
×	Multiplication
÷	Division (yielding quotient)
+	Two's-complement addition
−	Two's-complement subtraction, unary minus
=, ≠	Equals and Not Equals relations
<, ≤, >, ≥	Signed comparison relations
$<_u$, $\geq_u$	Unsigned comparison relations
?	Unordered comparison relation
&, |	AND, OR logical operators
⊕, ≡	Exclusive-OR, Equivalence logical operators ((a≡b) = (a⊕¬b))
CEIL(x)	Least integer ≥ x
DOUBLE(x)	Result of converting x from floating-point single format to floating-point double format, using the model shown on page 100
EXTS(x)	Result of extending x on the left with sign bits
GPR(x)	General Purpose Register x
MASK(x, y)	Mask having 1's in positions x through y (wrapping if x > y) and 0's elsewhere
MEM(x, y)	Contents of y bytes of memory starting at address x
$ROTL_{64}(x, y)$	Result of rotating the 64-bit value x left y positions
$ROTL_{32}(x, y)$	Result of rotating the 64-bit value x||x left y positions, where x is 32 bits long
SINGLE(x)	Result of converting x from floating-point double format to floating-point single format, using the model shown on page 103
SPREG(x)	Special Purpose Register x
TRAP	Invoke the system trap handler
characterization	Reference to the setting of status bits, in a standard way that is explained in the text
undefined	An undefined value. The value may vary from one implementation to another, and from one execution to another on the same implementation.
CIA	Current Instruction Address, which is the 64{32}-bit address of the instruction being described by a sequence of RTL. Used by relative branches to set the Next Instruction Address (NIA), and by *Branch* instructions with LK=1 to set the Link Register. In 32-bit mode of 64-bit implementations, the high-order 32 bits of CIA are always set to 0. Does not correspond to any architected register.
NIA	Next Instruction Address, which is the 64{32}-bit address of the next instruction to be executed. For a successful branch, the next instruction address is the branch target address: in RTL, this indicated by assigning a value to NIA. For other instructions that cause non-sequential instruction fetching (see Book III, *PowerPC Operating Environment Architecture*), the RTL is similar. For instructions that do not branch, and do not otherwise cause instruction fetching to be non-sequential, the next instruction address is CIA+4. In 32-bit mode of 64-bit implementations, the high-order 32 bits of NIA are always set to 0. Does not correspond to any architected register.
if ... then ... else ...	Conditional execution, indenting shows range, else is optional
do	Do loop, indenting shows range. "To" and/or "by" clauses specify incrementing an iteration variable, and "while" and/or "until" clauses give termination conditions, in the usual manner.
leave	Leave innermost do loop, or do loop described in leave statement

The precedence rules for RTL operators are summarized in Table 1 on page 5. Operators higher in the table are applied before those lower in the table. Operators at the same level in the table associate from left to right, from right to left, or not at all, as shown. (For example, − associates from left to right, so a−b−c = (a−b)−c.) Parentheses are used to override the evaluation order implied by the table, or to increase clarity: parenthesized expressions are evaluated before serving as operands.

Table 1. Operator Precedence	
Operators	**Associativity**
subscript, function evaluation	left to right
pre-superscript (replication), post-superscript (exponentiation)	*right to left*
unary –, ¬	*right to left*
×, ÷	left to right
+, –	left to right
‖	left to right
=, ≠, <, ≤, >, ≥, ≮, ≯, ?	left to right
&, ⊕, ≡	left to right
|	left to right
: (range)	none
←	none

1.6 Processor Overview

The processor implements the instruction set, the storage model, and other facilities defined in this document. Instructions which the processor can execute fall into the following classes.

- branch instructions,
- fixed-point instructions, and
- floating-point instructions.

Branch instructions are described in Section 2.4, "Branch Processor Instructions" on page 18. Fixed-point instructions are described in Section 3.3, "Fixed-Point Processor Instructions" on page 29. Floating-point instructions are described in Section 4.6, "Floating-Point Processor Instructions" on page 99.

Fixed-point instructions operate on byte, halfword, word, and, in 64-bit implementations, doubleword operands. Floating-point instructions operate on single-precision and double-precision floating-point operands. The PowerPC Architecture uses instructions that are four bytes long and word-aligned. It provides for byte, halfword, word, and, in 64-bit implementations, doubleword operand fetches and stores between storage and a set of 32 General Purpose Registers (GPRs). It also provides for word and doubleword operand fetches and stores between storage and a set of 32 Floating-Point Registers (FPRs).

There are no computational instructions that modify storage. To use a storage operand in a computation and then modify the same or another storage location, the content of storage must be loaded into a register, modified, and then stored back to the target location. Figure 1 is a logical representation of instruction processing. Figure 2 on page 6 shows the registers of the PowerPC User Instruction Set Architecture.

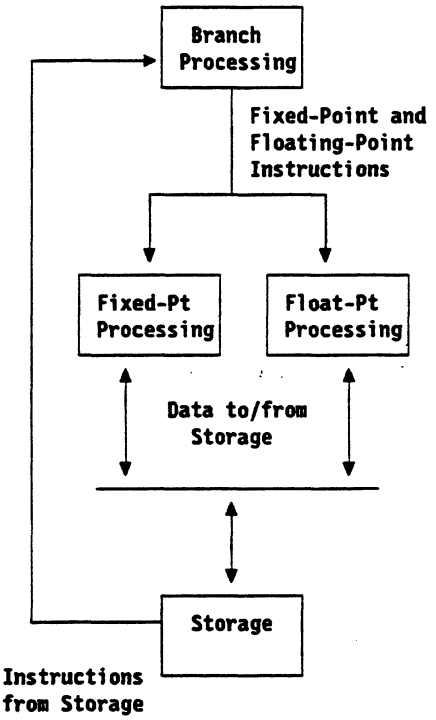


Figure 1. Logical Processing Model

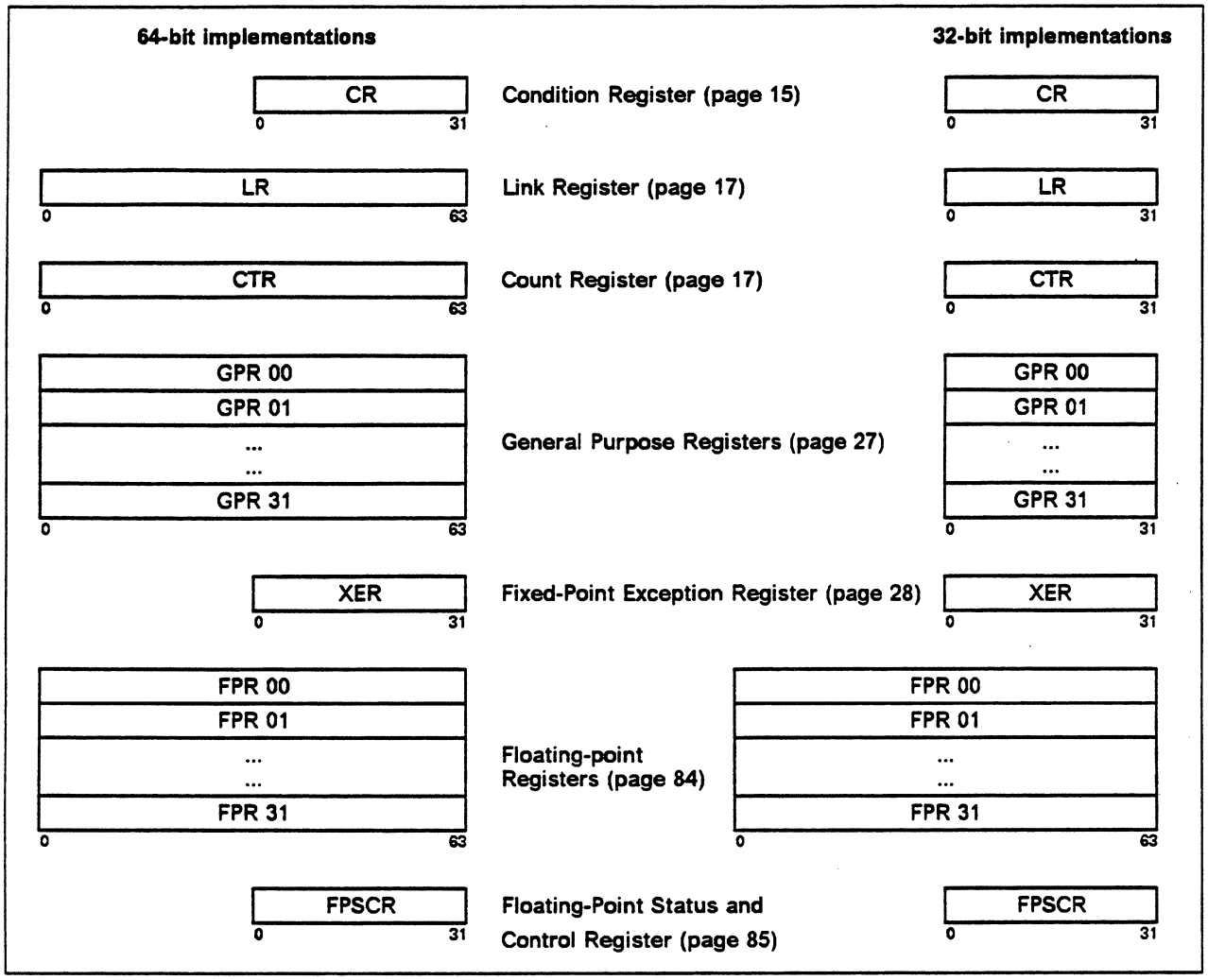


Figure 2. PowerPC User Register Set

1.7 Instruction Formats

All instructions are four bytes long and word-aligned. Thus, whenever instruction addresses are presented to the processor (as in *Branch* instructions) the two low order bits are ignored. Similarly, whenever the processor develops an instruction address its two low order bits are zero.

Bits 0:5 always specify the opcode (OPCD, below). Many instructions also have an extended opcode (XO, below). The remaining bits of the instruction contain one or more fields as shown below for the different instruction formats.

In some cases an instruction field is reserved, or must contain a particular value. These cases are not shown in the format diagrams given below, but are shown in the individual instruction layouts as appropriate. 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 and the results are as described in Section 1.9.2, "Invalid Instruction Forms" on page 11.

Split Field Notation

In some cases an instruction field occupies more than one contiguous sequence of bits, or occupies one contiguous sequence of bits which are used in permuted order. Such a field is called a "split field." In the format diagrams given below and in the individual instruction layouts, the name of a split field is shown in small letters, once for each of the contiguous sequences. In the RTL description of an instruction having a split field, and in certain other places where individual bits of a split field are identified, the name of the field in small letters represents the concatenation of the sequences from left to right. In all other places, the name of the field is capitalized, and represents the concatenation of the sequences in some order, which need not be left to right, as described for each affected instruction.

1.7.1 I Form

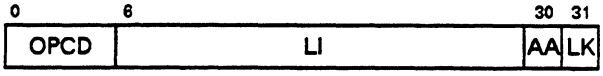


Figure 3. I Instruction Format

1.7.2 B Form

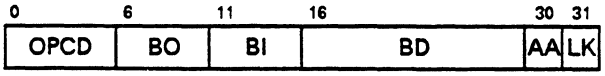


Figure 4. B Instruction Format

1.7.3 SC Form

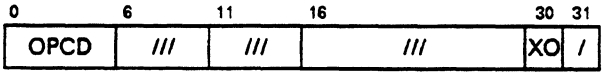


Figure 5. SC Instruction Format

1.7.4 D Form

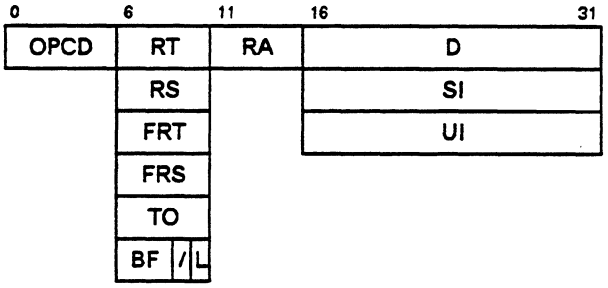


Figure 6. D Instruction Format

1.7.5 DS Form

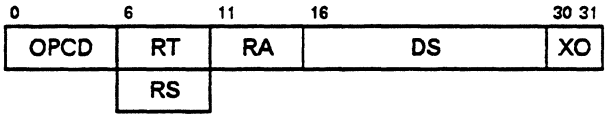


Figure 7. DS Instruction Format (64-bit implementations only)

1.7.6 X Forms

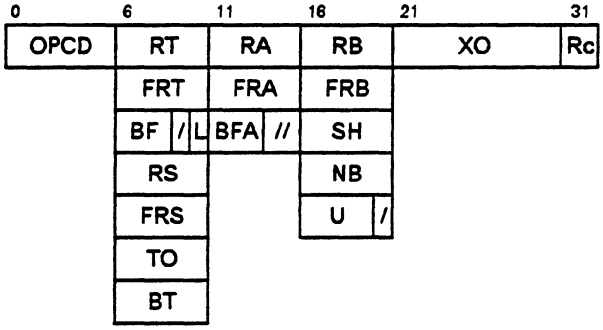


Figure 8. X Instruction Format

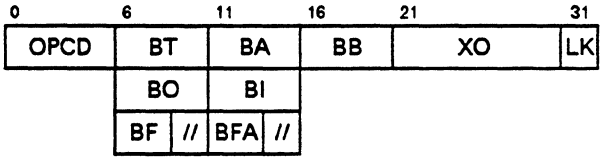


Figure 9. XL Instruction Format

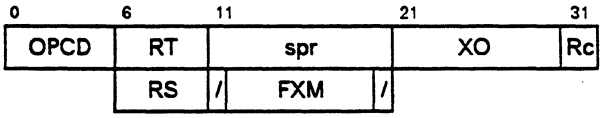


Figure 10. XFX Instruction Format

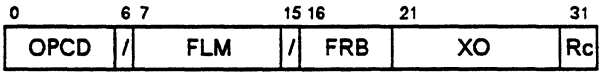


Figure 11. XFL Instruction Format

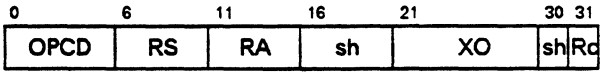


Figure 12. XS Instruction Format (64-bit implementations only)

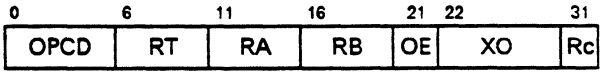


Figure 13. XO Instruction Format

1.7.7 A Form

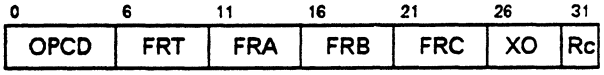


Figure 14. A Instruction Format

1.7.8 M Form

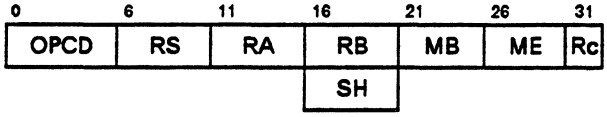


Figure 15. M Instruction Format

1.7.9 MD Form

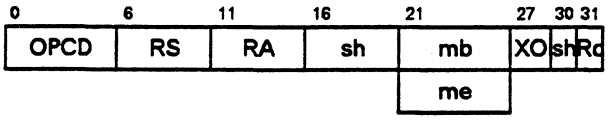


Figure 16. MD Instruction Format (64-bit implementations only)

1.7.10 MDS Form

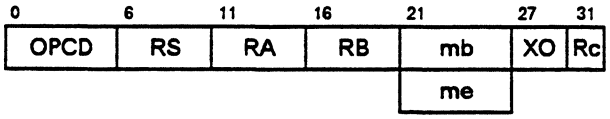


Figure 17. MDS Instruction Format (64-bit implementations only)

1.7.11 Instruction Fields

AA (30)
Absolute Address bit

0 The immediate field represents 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 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 and the address of the branch instruction.

1 The immediate field represents an absolute address. For I-form branches the effective address of the branch target is the LI field sign-extended to 64 bits. For B-form branches the effective address of the branch target is the BD field sign-extended to 64 bits.

BA (11:15)
Field used to specify a bit in the CR to be used as a source.

BB (16:20)
Field used to specify a bit in the CR to be used as a source.

BD (16:29)
Immediate field specifying a 14-bit signed two's complement branch displacement which is concatenated on the right with 0b00 and sign-extended to 64 bits.

BF (6:8)
Field used to specify one of the CR fields or one of the FPSCR fields as a target.

BFA (11:13)
Field used to specify one of the CR fields or one of the FPSCR fields as a source.

BI (11:15)
Field used to specify a bit in the CR to be used as the condition of a *Branch Conditional* instruction.

BO (6:10)
Field used to specify options for the *Branch Conditional* instructions. The encoding is described in Section 2.4, "Branch Processor Instructions" on page 18.

BT (6:10)
Field used to specify a bit in the CR or in the FPSCR as the target of the result of an instruction.

D (16:31)
Immediate field specifying a 16-bit signed two's complement integer which is sign-extended to 64 bits.

DS (16:29)
Immediate field specifying a 14-bit signed two's complement integer which is concatenated on the right with 0b00 and sign-extended to 64 bits. This field is defined in 64-bit implementations only.

FLM (7:14)
Field mask used to identify the FPSCR fields that are to be updated by the ***mtfsf*** instruction.

FRA (11:15)
Field used to specify an FPR as a source of an operation.

FRB (16:20)
Field used to specify an FPR as a source of an operation.

FRC (21:25)
Field used to specify an FPR as a source of an operation.

FRS (6:10)
Field used to specify an FPR as a source of an operation.

FRT (6:10)
Field used to specify an FPR as the target of an operation.

FXM (12:19)
Field mask used to identify the CR fields that are to be updated by the ***mtcrf*** instruction.

L (10)
Field used to specify whether a Fixed-Point *Compare* instruction is to compare 64-bit numbers or 32-bit numbers. This field is defined in 64-bit implementations only.

LI (6:29)
Immediate field specifying a 24-bit signed two's complement integer which is concatenated on the right with 0b00 and sign-extended to 64 bits.

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 into the Link Register.

MB (21:25) and ME (26:30)
Fields used in M-form instructions to specify a 64-bit mask consisting of 1-bits from bit MB+32 through bit ME+32 inclusive, and 0-bits elsewhere, as described in Section 3.3.13, "Fixed-Point Rotate and Shift Instructions" on page 69.

MB (21:26)
Field used in MD-form and MDS-form instructions to specify the first 1-bit of a 64-bit mask, as described in Section 3.3.13, "Fixed-Point Rotate and Shift Instructions" on page 69. This field is defined in 64-bit implementations only.

ME (21:26)
Field used in MD-form and MDS-form instructions to specify the last 1-bit of a 64-bit mask, as described in Section 3.3.13, "Fixed-Point Rotate and Shift Instructions" on page 69. This field is defined in 64-bit implementations only.

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

OPCD (0:5)
Primary opcode field.

OE (21)
Used for extended arithmetic to enable setting OV and SO in the XER.

RA (11:15)
Field used to specify a GPR to be used as a source or as a target.

RB (16:20)
Field used to specify a GPR to be used as a source.

Rc (31)
RECORD bit

0 Do not set the Condition Register.

1 Set the Condition Register 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. The result as an unsigned quantity or a bit string can be deduced 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)
Field used to specify a GPR to be used as a source.

RT (6:10)
Field used to specify a GPR to be used as a target.

SH (16:20, or 16:20 and 30)
Field used to specify a shift amount. Location 16:20 and 30 pertains to 64-bit implementations only.

SI (16:31)
Immediate field used to specify a 16-bit signed integer.

SPR (11:20)
Field used to specify a Special Purpose Register for the ***mtspr*** and ***mfspr*** instructions. The encoding is described in Section 3.3.14, "Move To/From System Register Instructions" on page 79.

TO (6:10)
Field used to specify the conditions on which to trap. The encoding is described in Section 3.3.11, "Fixed-Point Trap Instructions" on page 61.

U (16:19)
Immediate field used as the data to be placed into a field in the FPSCR.

UI (16:31)
Immediate field used to specify a 16-bit unsigned integer.

XO (21:29, 21:30, 22:30, 26:30, 27:29, 27:30, 30, or 30:31)
Extended opcode field. Locations 21:29, 27:29, 27:30, and 30:31 pertain to 64-bit implementations only.

1.8 Classes of Instructions

An instruction falls into exactly one of the following three classes:

Defined
Illegal
Reserved

The class is determined by examining the opcode, and the extended opcode if any. If the opcode, or combination of opcode and extended opcode, is not that of a defined instruction nor of a reserved instruction, the instruction is illegal.

Some instructions are defined only for 64-bit implementations and a few are defined only for 32-bit implementations (see 1.8.2, "Illegal Instruction Class"). With the exception of these, a given instruction is in the same class for all implementations of the PowerPC Architecture. In future versions of this architecture, instructions that are now illegal may become defined (by being added to the architecture) or reserved (by being assigned to one of the special purposes described in Appendix J, "Reserved Instructions" on page 177). Similarly, instructions that are now reserved may become defined.

The results of attempting to execute a given instruction are said to be *boundedly undefined* if they could have been achieved by executing an arbitrary sequence of defined instructions, in valid form (see below), starting in the state the machine was in before attempting to execute the given instruction. Boundedly undefined results for a given instruction may vary between implementations, and between execution attempts in the same implementation, and are not further defined in this document.

1.8.1 Defined Instruction Class

This class of instructions contains all the instructions defined in the PowerPC User Instruction Set Architecture, PowerPC Virtual Environment Architecture, and PowerPC Operating Environment Architecture.

Defined instructions are guaranteed to be supported in all implementations, except as stated in the instruction descriptions. (The exceptions are instructions that are supported only in 64-bit implementations or only in 32-bit implementations.)

A defined instruction can have preferred and/or invalid forms, as described in Section 1.9.1, "Preferred Instruction Forms" on page 11, and Section 1.9.2, "Invalid Instruction Forms" on page 11.

1.8.2 Illegal Instruction Class

This class of instructions contains the set of instructions described in Appendix I, "Illegal Instructions" on page 175. For 64-bit implementations this class includes all instructions that are defined only for 32-bit implementations. For 32-bit implementations it includes all instructions that are defined only for 64-bit implementations.

Excluding instructions that are defined for one type of implementation but not the other, illegal instructions are available for future extensions of the PowerPC Architecture: that is, some future version of the PowerPC Architecture may define any of these instructions to perform new functions.

Any attempt to execute an illegal instruction will cause the system illegal instruction error handler to be invoked and will have no other effect.

An instruction consisting entirely of binary 0's is guaranteed always to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized storage will result in the invocation of the system illegal instruction error handler.

Editors' Note

Instructions in this class were formerly called "invalid instructions." The term was changed to "illegal instructions" to reduce confusion between these instructions and invalid *forms* of *defined* instructions.

1.8.3 Reserved Instruction Class

This class of instructions contains the set of instructions described in Appendix J, "Reserved Instructions" on page 177.

Reserved instructions are allocated to specific purposes that are outside the scope of the PowerPC Architecture.

Any attempt to execute a reserved instruction will either cause the system illegal instruction error handler to be invoked or will yield boundedly undefined results.

Engineering Note

Causing the system illegal instruction error handler to be invoked if attempt is made to execute a reserved instruction, that is not defined in Book IV, *PowerPC Implementation Features* for the implementation, facilitates the debugging of software.

1.9 Forms of Defined Instructions

1.9.1 Preferred Instruction Forms

Some of the defined instructions have preferred forms. For such an instruction, the preferred form will execute in an efficient manner, but any other form may take significantly longer to execute than the preferred form.

Instructions having preferred forms are:

- the *Load/Store Multiple* instructions
- the *Load/Store String* instructions
- the *Or Immediate* instruction (preferred form of no-op)

1.9.2 Invalid Instruction Forms

Some of the defined instructions have invalid forms. An instruction form is invalid if one or more fields of the instruction, excluding the opcode field(s), are coded incorrectly.

Any attempt to execute an invalid form of an instruction will either cause the system illegal instruction error handler to be invoked or will yield boundedly undefined results. Exceptions to this rule are stated in the instruction descriptions.

Some kinds of invalid form can be deduced from the instruction layout. These are listed below.

- Rc bit shown as '/' but coded as 1, or shown as 1 but coded as 0.
- LK bit shown as '/' but coded as 1.
- OE bit shown as '/' but coded as 1.
- Other field shown as '/'(s) but coded as non-zero.

These invalid forms are not discussed further.

Instructions having invalid forms that cannot be so deduced are listed below. For these, the invalid forms are identified in the instruction descriptions.

- the *Branch Conditional* instructions
- the *Load/Store with Update* instructions
- the *Load Multiple* instructions
- the *Load String* instructions
- the *Fixed-Point Compare* instructions (invalid form exists only in 32-bit implementations)
- *Move To/From Special Purpose Register* (***mtspr***, ***mfspr***)
- the *Load/Store Floating-Point with Update* instructions

Assembler Note

To the extent possible, the Assembler should report uses of invalid instruction forms as errors.

Engineering Note

Causing the system illegal instruction error handler to be invoked if attempt is made to execute an invalid form of an instruction facilitates the debugging of software.

1.9.3 Optional Instructions

Some of the defined instructions are optional.

Any attempt to execute an optional instruction that is not provided by the implementation will cause the system illegal instruction error handler to be invoked. Exceptions to this rule are stated in the instruction descriptions.

1.10 Exceptions

There are two kinds of exception, those caused directly by the execution of an instruction and those caused by an asynchronous event. In either case, the exception may cause one of several components of the system software to be invoked.

The exceptions that can be caused directly by the execution of an instruction are the following.

- an attempt to execute an illegal instruction, or an attempt by an application program to execute a "privileged" instruction (see Book III, *PowerPC Operating Environment Architecture*) (system illegal instruction error handler or system privileged instruction error handler)
- the execution of a defined instruction using an invalid form (system illegal instruction error handler or system privileged instruction error handler)
- the execution of an optional instruction that is not provided by the implementation (system illegal instruction error handler)
- an attempt to access a storage location that is unavailable (system data storage error handler or system instruction storage error handler)
- an attempt to access storage in a manner that violates storage protection (system data storage error handler or system instruction storage error handler)
- an attempt to access storage with an effective address alignment that is invalid for the instruction (system alignment error handler)

- the execution of a *System Call* instruction (system service program)
- the execution of a *Trap* instruction that traps (system trap handler)
- the execution of a floating-point instruction when floating-point instructions are unavailable (system floating-point unavailable error handler)
- the execution of a floating-point instruction that causes a floating-point exception that is enabled (system floating-point enabled exception error handler)
- the execution of a floating-point instruction that requires system software assistance (system floating-point assist error handler; the conditions under which such software assistance is required are implementation-dependent)

The exceptions that can be caused by an asynchronous event are described in Book III, *PowerPC Operating Environment Architecture*.

The invocation of the system error handler is precise, except that if one of the imprecise modes for invoking the system floating-point enabled exception error handler is in effect (see page 92) then the invocation of the system floating-point enabled exception error handler may be imprecise. When the system error handler is invoked imprecisely, the excepting instruction does not appear to complete before the next instruction starts (because one of the effects of the excepting instruction, namely the invocation of the system error handler, has not yet occurred).

Additional information about exception handling can be found in Book III, *PowerPC Operating Environment Architecture*.

1.11 Storage Addressing

A program references storage using the effective address computed by the processor when it executes a *Storage Access* or *Branch* instruction (or certain other instructions described in Book II, *PowerPC Virtual Environment Architecture*, and Book III, *PowerPC Operating Environment Architecture*), or when it fetches the next sequential instruction.

1.11.1 Storage Operands

Bytes in storage are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

Storage operands may be bytes, halfwords, words, or doublewords, or, for the *Load/Store Multiple* and *Move Assist* instructions, a sequence of bytes or words. The address of a storage operand is the address of its first byte (i.e., of its lowest-numbered byte). Byte ordering is Big-Endian by default, but PowerPC can be operated in a mode in which byte ordering is Little-Endian. See Appendix D, "Little-Endian Byte Ordering" on page 145.

Operand length is implicit for each instruction.

The operand of a single-register *Storage Access* instruction has a "natural" alignment boundary equal to the operand length. In other words, the "natural" address of an operand is an integral multiple of the operand length. A storage operand is said to be "aligned" if it is aligned at its natural boundary: otherwise it is said to be "unaligned."

Storage operands for single-register *Storage Access* instructions have the following characteristics. (Although not permitted as storage operands, quadwords are shown because quadword alignment is desirable for certain storage operands.)

Operand	Length	$Addr_{60:63}$ if aligned
Byte	8 bits	xxxx
Halfword	2 bytes	xxx0
Word	4 bytes	xx00
Doubleword	8 bytes	x000
Quadword	16 bytes	0000

Note: An "x" in an address bit position indicates that the bit can be 0 or 1 independent of the state of other bits in the address.

The concept of alignment is also applied more generally, to any datum in storage. For example, a 12-byte datum in storage is said to be word-aligned if its address is an integral multiple of 4.

Some instructions require their storage operands to have certain alignments. In addition, alignment may affect performance. For single-register *Storage Access* instructions the best performance is obtained when storage operands are aligned. Additional effects of data placement on performance are described in Book II, *PowerPC Virtual Environment Architecture*.

Instructions are always four bytes long and word-aligned.

1.11.2 Effective Address Calculation

The 64- or 32-bit address computed by the processor when executing a *Storage Access* or *Branch* instruction (or certain other instructions described in Book II, *PowerPC Virtual Environment Architecture*, and Book III, *PowerPC Operating Environment Architecture*), or when fetching the next sequential instruction, is called the "effective address," and specifies a byte in storage. For a *Storage Access* instruction, if the sum of the effective address and the operand length

exceeds the maximum effective address, the storage operand is considered to wrap around from the maximum effective address to effective address 0, as described below.

Effective address computations, for both data and instruction accesses, use 64{32}-bit unsigned binary arithmetic regardless of mode. A carry from bit 0 is ignored. In a 64-bit implementation, the 64-bit current instruction address and next instruction address are not affected by a change from 32-bit mode to 64-bit mode, but they are affected by a change from 64-bit mode to 32-bit mode (the high-order 32 bits are set to 0).

In 64-bit mode, the entire 64-bit result comprises the 64-bit effective address. The effective address arithmetic wraps around from the maximum address, $2^{64}-1$, to address 0.

In 32-bit mode, the low-order 32 bits of the 64-bit result comprise the effective address for the purpose of addressing storage. The high-order 32 bits of the 64-bit effective address are ignored for the purpose of accessing data, but are included whenever a 64-bit effective address is placed into a GPR by *Load with Update* and *Store with Update* instructions. The high-order 32 bits of the 64-bit effective address are set to 0 for the purpose of fetching instructions, and whenever a 64-bit effective address is placed into the Link Register by *Branch* instructions having LK=1. The high-order 32 bits of the 64-bit effective address are set to 0 in Special Purpose Registers when the system error handler is invoked. As used to address storage, the effective address arithmetic appears to wrap around from the maximum address, $2^{32}-1$, to address 0.

A zero in the RA field indicates the absence of the corresponding address component. For the absent component, a value of zero is used for the address. This is shown in the instruction descriptions as (RA|0).

In both 64-bit and 32-bit modes, the calculated Effective Address may be modified in its three low-order bits before accessing storage if the PowerPC system is operating in Little-Endian mode. See Appendix D, "Little-Endian Byte Ordering" on page 145.

Effective addresses are computed as follows. In the descriptions below, it should be understood that "the contents of a GPR" refers to the entire 64-bit contents, independent of mode, but that in 32-bit mode, only bits 32:63 of the 64-bit result of the computation are used to address storage.

- With X-form instructions, in computing the effective address of a data element, the contents of the GPR designated by RB is added to the contents of the GPR designated by RA or to zero if RA=0.
- With D-form instructions, the 16-bit D field is sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.
- With DS-form instructions, the 14-bit DS field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.
- With I-form *Branch* instructions, the 24-bit LI field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. If AA=0, this address component is added to the address of the branch instruction to form the effective address of the next instruction. If AA=1, this address component is the effective address of the next instruction.
- With B-form *Branch* instructions, the 14-bit BD field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. If AA=0, this address component is added to the address of the branch instruction to form the effective address of the next instruction. If AA=1, this address component is the effective address of the next instruction.
- With XL-form *Branch* instructions, bits 0:61 of the Link Register or the Count Register are concatenated on the right with 0b00 to form the effective address of the next instruction.
- With sequential instruction fetching, the value 4 is added to the address of the current instruction to form the effective address of the next instruction.

Chapter 2. Branch Processor

2.1 Branch Processor Overview

This chapter describes the registers and instructions that make up the Branch Processor facilities. Section 2.3, "Branch Processor Registers" on page 15 describes the registers associated with the Branch Processor. Section 2.4, "Branch Processor Instructions" on page 18 describes the instructions associated with the Branch Processor.

2.2 Instruction Fetching

In general, instructions appear to execute sequentially, in the order in which they appear in storage. The exceptions to this rule are listed below.

- *Branch* instructions for which the branch is taken cause execution to continue at the target address generated by the *Branch* instruction.
- *Trap* and *System Call* instructions cause the appropriate system handler to be invoked.
- Exceptions can cause the system error handler to be invoked, as described in Section 1.10, "Exceptions" on page 11.
- The *Return From Interrupt* instruction, described in Book III, *PowerPC Operating Environment Architecture*, causes execution to continue at the address contained in a Special Purpose Register.

In general, each instruction appears to complete before the next instruction starts. The only exceptions to this rule arise when the system error handler is invoked imprecisely, as described in Section 1.10, "Exceptions" on page 11, or when certain special registers are altered, as described in the appendix entitled "Synchronization Requirements for Special Registers" in Book III, *PowerPC Operating Environment Architecture* (none of these special registers can be altered by an application program).

Programming Note

CAUTION
Implementations are allowed to prefetch any number of instructions before the instructions are actually executed. If a program modifies the instructions it intends to execute, it should call a system library program to ensure that the modifications have been made visible to the instruction fetching mechanism prior to attempting to execute the modified instructions.

2.3 Branch Processor Registers

2.3.1 Condition Register

The Condition Register (CR) is a 32-bit register which reflects the result of certain operations, and provides a mechanism for testing (and branching).

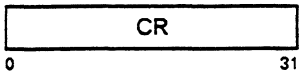


Figure 18. Condition Register

The bits in the Condition Register are grouped into eight 4-bit fields, named CR Field 0 (CR0), ..., CR Field 7 (CR7), which are set in one of the following ways:

- Specified fields of the CR can be set by a move to the CR from a GPR (***mtcrf***).
- A specified field of the CR can be set by a move to the CR from the another CR field (***mcrf***), from the XER (***mcrxr***), or from the FPSCR (***mcrfs***).
- CR Field 0 can be set as the implicit result of a fixed-point operation.
- CR Field 1 can be set as the implicit result of a floating-point operation.
- A specified CR field can be set as the result of either a fixed-point or a floating-point *Compare* instruction.

Instructions are provided to perform logical operations on individual CR bits, and to test individual CR bits.

When Rc = 1 in most fixed-point instructions, CR Field 0 (bits 0:3 of the Condition Register) is set by an algebraic comparison of the result (the low-order 32 bits of the result in 32-bit mode) to zero. ***addic.***, ***andi.***, and ***andis.*** set these four bits implicitly. These bits are interpreted as follows. As used below, "result" refers to the entire 64-bit value placed into the target register in 64-bit mode, and to bits 32:63 of the 64-bit value placed into the target register in 32-bit mode. If any portion of the result is undefined, then the value placed into CR Field 0 is undefined.

Bit Description

0 ***Negative (LT)***
The result is negative.

1 ***Positive (GT)***
The result is positive.

2 ***Zero (EQ)***
The result is zero.

3 ***Summary Overflow (SO)***
This is a copy of the final state of XER_{SO} at the completion of the instruction.

When Rc = 1 in all floating-point instructions except floating-point *Compare*, CR Field 1 (bits 4:7 of the Condition Register) is set to the Floating-Point exception status, copied from bits 0:3 of the Floating-Point Status and Control Register. These bits are interpreted as follows.

Bit Description

4 ***Floating-Point Exception (FX)***
This is a copy of the final state of $FPSCR_{FX}$ at the completion of the instruction.

5 ***Floating-Point Enabled Exception (FEX)***
This is a copy of the final state of $FPSCR_{FEX}$ at the completion of the instruction.

6 ***Floating-Point Invalid Operation Exception (VX)***
This is a copy of the final state of $FPSCR_{VX}$ at the completion of the instruction.

7 ***Floating-Point Overflow Exception (OX)***
This is a copy of the final state of $FPSCR_{OX}$ at the completion of the instruction.

When a specified CR field is set by a *Compare* instruction, the bits of the specified field are interpreted as follows.

Bit Description

0 ***Less Than, Floating-Point Less Than (LT, FL)***
For fixed-point *Compare* instructions, (RA) < SI, UI, or (RB) (algebraic comparison) or (RA) $\lneq$ SI, UI, or (RB) (logical comparison). For floating-point *Compare* instructions, (FRA) < (FRB).

1 ***Greater Than, Floating-Point Greater Than (GT, FG)***
For fixed-point *Compare* instructions, (RA) > SI, UI, or (RB) (algebraic comparison) or (RA) $\gneq$ SI, UI, or (RB) (logical comparison). For floating-point *Compare* instructions, (FRA) > (FRB).

2 ***Equal, Floating-Point Equal (EQ, FE)***
For fixed-point *Compare* instructions, (RA) = SI, UI, or (RB). For floating-point *Compare* instructions, (FRA) = (FRB).

3 ***Summary Overflow, Floating-Point Unordered (SO, FU)***
For fixed-point *Compare* instructions, this is a copy of the final state of XER_{SO} at the completion of the instruction. For floating-point *Compare* instructions, one or both of (FRA) and (FRB) is a NaN.

2.3.2 Link Register

The Link Register (LR) is a 64-bit register. It can be used to provide the branch target address for the *Branch Conditional to Link Register* instruction, and it holds the return address after *Branch and Link* instructions.

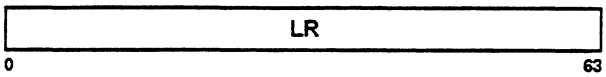


Figure 19. Link Register

2.3.3 Count Register

The Count Register (CTR) is a 64-bit register. It can be used to hold a loop count that can be decremented during execution of *Branch* instructions that contain an appropriately coded BO field. If the value in the Count Register is 0 before being decremented, it is −1 afterward. The Count Register can also be used to provide the branch target address for the *Branch Conditional to Count Register* instruction.

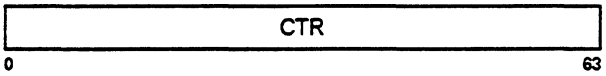


Figure 20. Count Register

2.4 Branch Processor Instructions

2.4.1 Branch Instructions

The sequence of instruction execution can be changed by the *Branch* instructions. Because all instructions are on word boundaries, bits 62 and 63 of the generated branch target address are ignored by the processor in performing the branch.

The *Branch* instructions compute the effective address (EA) of the target in one of the following four ways, as described in Section 1.11.2, "Effective Address Calculation" on page 12.

1. Adding a displacement to the address of the branch instruction (*Branch* or *Branch Conditional* with AA=0).
2. Specifying an absolute address (*Branch* or *Branch Conditional* with AA=1).
3. Using the address contained in the Link Register (*Branch Conditional to Link Register*).
4. Using the address contained in the Count Register (*Branch Conditional to Count Register*).

In all four cases, in 32-bit mode of 64-bit implementations, the final step in the address computation is setting the high-order 32 bits of the target address to 0.

For the first two methods, the target addresses can be computed sufficiently ahead of the branch instruction that instructions can be prefetched along the target path. For the third and fourth methods, prefetching instructions along the target path is also possible provided the Link Register or the Count Register is loaded sufficiently ahead of the branch instruction.

Branching can be conditional or unconditional, and the return address can optionally be provided. If the return address is to be provided (LK=1), the effective address of the instruction following the branch instruction is placed into the Link Register after the branch target address has been computed: this is done whether or not the branch is taken.

In *Branch Conditional* instructions, the BO field specifies the conditions under which the branch is taken. The first four bits of the BO field specify how the branch is affected by or affects the Condition Register and the Count Register. The fifth bit, shown below as having the value "y," may be used by some implementations as described below.

The encoding for the BO field is as follows. Here $M=32$ in 32-bit mode and $M=0$ in 64-bit mode. If the BO field specifies that the CTR is to be decremented, the entire 64-bit CTR is decremented regardless of the mode.

BO	Description
0000y	Decrement the CTR, then branch if the decremented $CTR_{M:63} \neq 0$ and the condition is FALSE.
0001y	Decrement the CTR, then branch if the decremented $CTR_{M:63} = 0$ and the condition is FALSE.
001zy	Branch if the condition is FALSE.
0100y	Decrement the CTR, then branch if the decremented $CTR_{M:63} \neq 0$ and the condition is TRUE.
0101y	Decrement the CTR, then branch if the decremented $CTR_{M:63} = 0$ and the condition is TRUE.
011zy	Branch if the condition is TRUE.
1z00y	Decrement the CTR, then branch if the decremented $CTR_{M:63} \neq 0$.
1z01y	Decrement the CTR, then branch if the decremented $CTR_{M:63} = 0$.
1z1zz	Branch always.

Above, "z" denotes a bit that must be zero: if it is not zero the instruction form is invalid.

The "y" bit provides a hint about whether a conditional branch is likely to be taken, and may be used by some implementations to improve performance.

The "branch always" encoding of the BO field does not have a "y" bit.

For *Branch Conditional* instructions that have a "y" bit, using y=0 indicates that the following behavior is likely.

- If the instruction is ***bc[l][a]*** with a negative value in the displacement field, the branch is taken.
- In all other cases (***bc[l][a]*** with a non-negative value in the displacement field, ***bclr[l]***, or ***bcctr[l]***), the branch falls through (is not taken).

Using y=1 reverses the preceding indications.

The displacement field is used as described above even if the target is an absolute address.

Programming Note

The default value for the "y" bit should be 0: the value 1 should be used only if software has determined that the prediction corresponding to y=1 is more likely to be correct than the prediction corresponding to y=0.

Engineering Note

For all three *Branch Conditional* instructions, the branch should be predicted to be taken if the value of the following expression is 1, and to fall through if the value is 0.

$$((BO_0 \ \& \ BO_2) \mid s) \oplus BO_4$$

Here "s" is bit 16 of the instruction, which is the sign bit of the displacement field if the instruction has a displacement field and is 0 otherwise. BO_4 is the "y" bit, or 0 for the "branch always" encoding of the BO field. (Advantage is taken of the fact that, for ***bclr[l]*** and ***bcctr[l]***, bit 16 of the instruction is part of a reserved field and therefore must be 0.)

Extended mnemonics for branches

Many extended mnemonics are provided so that *Branch Conditional* instructions can be coded with the condition as part of the instruction mnemonic rather than as a numeric operand. Some of these are shown as examples with the *Branch* instructions. See Appendix C, "Assembler Extended Mnemonics" on page 133 for additional extended mnemonics.

Programming Note

In some implementations the processor may keep a stack of the Link Register values most recently set by *Branch and Link* instructions, with the possible exception of the form shown below for obtaining the address of the next instruction. To benefit from this stack, the following programming conventions should be used.

Let A, B, and Glue be programs.

- Obtaining the address of the next instruction:
  Use the following form of *Branch and Link*.

  ```
  bcl   20,31,$+4
  ```

- Loop counts:
  Keep them in the Count Register, and use one of the *Branch Conditional* instructions to decrement the count and to control branching (e.g., branching back to the start of a loop if the decremented counter value is non-zero).

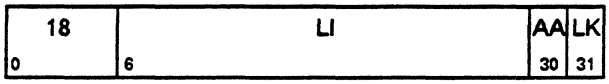
- Computed goto's, case statements, etc.:
  Use the Count Register to hold the address to branch to, and use the ***bcctr*** instruction (LK=0) to branch to the selected address.

- Direct subroutine linkage:
  Here A calls B and B returns to A. The two branches should be as follows.
  - A calls B: use a *Branch* instruction that sets the Link Register (LK=1).
  - B returns to A: use the ***bclr*** instruction (LK=0) (the return address is in, or can be restored to, the Link Register).

- Indirect subroutine linkage:
  Here A calls Glue, Glue calls B, and B returns to A rather than to Glue. (Such a calling sequence is common in linkage code used when the subroutine that the programmer wants to call, here B, is in a different module from the caller: the Binder inserts "glue" code to mediate the branch.) The three branches should be as follows.
  - A calls Glue: use a *Branch* instruction that sets the Link Register (LK=1).
  - Glue calls B: place the address of B in the Count Register, and use the ***bcctr*** instruction (LK=0).
  - B returns to A: use the ***bclr*** instruction (LK=0) (the return address is in, or can be restored to, the Link Register).

Branch I-form

b	target_addr	(AA=0 LK=0)
ba	target_addr	(AA=1 LK=0)
bl	target_addr	(AA=0 LK=1)
bla	target_addr	(AA=1 LK=1)




```
if AA then NIA ← EXTS(LI || 0b00)
else       NIA ← CIA + EXTS(LI || 0b00)
if LK then
  LR ← CIA + 4
```

target_addr specifies the branch target address.

If AA=0 then the branch target address is the sum of LI || 0b00 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode of 64-bit implementations.

If AA=1 then the branch target address is the value LI || 0b00 sign-extended, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode of 64-bit implementations.

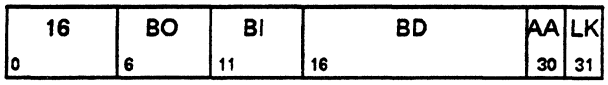
If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

LR	(if LK=1)

Branch Conditional B-form

bc	BO,BI,target_addr	(AA=0 LK=0)
bca	BO,BI,target_addr	(AA=1 LK=0)
bcl	BO,BI,target_addr	(AA=0 LK=1)
bcla	BO,BI,target_addr	(AA=1 LK=1)




```
if (64-bit implementation) & (64-bit mode)
  then M ← 0
  else M ← 32
if ¬BO2 then CTR ← CTR - 1
ctr_ok ← BO2 | ((CTRM:63 ≠ 0) ⊕ BO3)
cond_ok ← BO0 | (CRBI ≡ BO1)
if ctr_ok & cond_ok then
  if AA then NIA ← EXTS(BD || 0b00)
  else       NIA ← CIA + EXTS(BD || 0b00)
if LK then
  LR ← CIA + 4
```

The BI field specifies the bit in the Condition Register to be used as the condition of the branch. The BO field is used as described above. *target_addr* specifies the branch target address.

If AA=0 then the branch target address is the sum of BD || 0b00 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode of 64-bit implementations.

If AA=1 then the branch target address is the value BD || 0b00 sign-extended, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode of 64-bit implementations.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

CTR	(if BO_2=0)
LR	(if LK=1)

Extended Mnemonics:

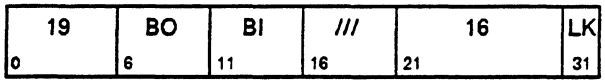
Examples of extended mnemonics for *Branch Conditional*:

Extended:		*Equivalent to:*	
blt	target	bc	12,0,target
bne	cr2,target	bc	4,10,target
bdnz	target	bc	16,0,target

Branch Conditional to Link Register XL-form

bclr	BO,BI	(LK=0)
bclrl	BO,BI	(LK=1)

[Power mnemonics: bcr, bcrl]




```
if (64-bit implementation) & (64-bit mode)
  then M ← 0
  else M ← 32
if ¬BO2 then CTR ← CTR - 1
ctr_ok ← BO2 | ((CTRM:63 ≠ 0) ⊕ BO3)
cond_ok ← BO0 | (CRBI ≡ BO1)
if ctr_ok & cond_ok then
  NIA ← LR0:61 || 0b00
if LK then
  LR ← CIA + 4
```

The BI field specifies the bit in the Condition Register to be used as the condition of the branch. The BO field is used as described above, and the branch target address is $LR_{0:61}$ || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode of 64-bit implementations.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

CTR	(if BO_2=0)
LR	(if LK=1)

Extended Mnemonics:

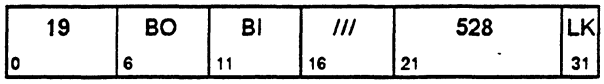
Examples of extended mnemonics for *Branch Conditional To Link Register*:

Extended:	*Equivalent to:*
bltlr	bclr 12,0
bnelr cr2	bclr 4,10
bdnzlr	bclr 16,0

Branch Conditional to Count Register XL-form

bcctr	BO,BI	(LK=0)
bcctrl	BO,BI	(LK=1)

[Power mnemonics: bcc, bccl]




```
cond_ok ← BO0 | (CRBI ≡ BO1)
if cond_ok then
  NIA ← CTR0:61 || 0b00
if LK then
  LR ← CIA + 4
```

The BI field specifies the bit in the Condition Register to be used as the condition of the branch. The BO field is used as described above, and the branch target address is $CTR_{0:61}$ || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode of 64-bit implementations.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

If the "decrement and test CTR" option is specified (BO_2=0), the instruction form is invalid.

Special Registers Altered:

LR	(if LK=1)

Extended Mnemonics:

Examples of extended mnemonics for *Branch Conditional To Count Register*:

Extended:	*Equivalent to:*
bltctr	bcctr 12,0
bnectr cr2	bcctr 4,10

2.4.2 System Call Instruction

This instruction provides the means by which a program can call upon the system to perform a service.

System Call SC-form

sc

[Power mnemonic: svca]

17	///	///	///	1	/
0	6	11	16	30	31

This instruction calls the system to perform a service. A complete description of this instruction can be found in Book III, *PowerPC Operating Environment Architecture.*

When control is returned to the program that executed the *System Call*, the content of the registers will depend on the register conventions used by the program providing the system service.

This instruction is context synchronizing (see Book III, *PowerPC Operating Environment Architecture*).

Special Registers Altered:

Dependent on the system service

Compatibility Note

For a discussion of Power compatibility with respect to instruction bits 16:29, please refer to Appendix G, "Incompatibilities with the Power Architecture" on page 165. For compatibility with future versions of this architecture, these bits should be coded as zero.

2.4.3 Condition Register Logical Instructions

Extended mnemonics for Condition Register logical operations

A set of extended mnemonics is provided that allow additional Condition Register logical operations, beyond those provided by the basic *Condition Register Logical* instructions, to be coded easily. Some of these are shown as examples with the *CR. Logical* instructions. See Appendix C, "Assembler Extended Mnemonics" on page 133 for additional extended mnemonics.

Condition Register AND XL-form

crand BT,BA,BB

19	BT	BA	BB	257	/
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA}$ & CR_{BB}

The bit in the Condition Register specified by BA is ANDed with the bit in the Condition Register specified by BB and the result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:
CR

Condition Register OR XL-form

cror BT,BA,BB

19	BT	BA	BB	449	/
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \mid CR_{BB}$

The bit in the Condition Register specified by BA is ORed with the bit in the Condition Register specified by BB and the result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:
CR

Extended Mnemonics:

Example of extended mnemonics for *Condition Register OR*:

Extended:	*Equivalent to:*
crmove Bx,By	cror Bx,By,By

Condition Register XOR XL-form

crxor BT,BA,BB

19	BT	BA	BB	193	/
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \oplus CR_{BB}$

The bit in the Condition Register specified by BA is XORed with the bit in the Condition Register specified by BB and the result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:
CR

Extended Mnemonics:

Example of extended mnemonics for *Condition Register XOR*:

Extended:	*Equivalent to:*
crclr Bx	crxor Bx,Bx,Bx

Condition Register NAND XL-form

crnand BT,BA,BB

19	BT	BA	BB	225	/
0	6	11	16	21	31

$CR_{BT} \leftarrow \neg(CR_{BA}$ & $CR_{BB})$

The bit in the Condition Register specified by BA is ANDed with the bit in the Condition Register specified by BB and the complemented result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:
CR

Condition Register NOR XL-form

crnor BT,BA,BB

19	BT	BA	BB	33	/
0	6	11	16	21	31

$CR_{BT} \leftarrow \neg(CR_{BA} \mid CR_{BB})$

The bit in the Condition Register specified by BA is ORed with the bit in the Condition Register specified by BB and the complemented result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:
CR

Extended Mnemonics:

Example of extended mnemonics for *Condition Register NOR*:

Extended:	*Equivalent to:*
crnot Bx,By	crnor Bx,By,By

Condition Register Equivalent XL-form

creqv BT,BA,BB

19	BT	BA	BB	289	/
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \equiv CR_{BB}$

The bit in the Condition Register specified by BA is XORed with the bit in the Condition Register specified by BB and the complemented result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:
CR

Extended Mnemonics:

Example of extended mnemonics for *Condition Register Equivalent*:

Extended:	*Equivalent to:*
crset Bx	creqv Bx,Bx,Bx

Condition Register AND With Complement XL-form

crandc BT,BA,BB

19	BT	BA	BB	129	/
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \,\&\, \neg CR_{BB}$

The bit in the Condition Register specified by BA is ANDed with the complement of the bit in the Condition Register specified by BB and the result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:
CR

Condition Register OR With Complement XL-form

crorc BT,BA,BB

19	BT	BA	BB	417	/
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \mid \neg CR_{BB}$

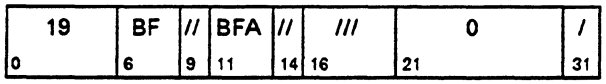
The bit in the Condition Register specified by BA is ORed with the complement of the bit in the Condition Register specified by BB and the result is placed into the bit in the Condition Register specified by BT.

Special Registers Altered:
CR

2.4.4 Condition Register Field Instruction

Move Condition Register Field XL-form

mcrf BF,BFA



$CR_{4\times BF:4\times BF+3} \leftarrow CR_{4\times BFA:4\times BFA+3}$

The contents of Condition Register field BFA are copied into Condition Register field BF.

Special Registers Altered:

CR

Chapter 3. Fixed-Point Processor

3.1 Fixed-Point Processor Overview

This chapter describes the registers and instructions that make up the Fixed-Point Processor facility. Section 3.2, "Fixed-Point Processor Registers" on page 27 describes the registers associated with the Fixed-Point Processor. Section 3.3, "Fixed-Point Processor Instructions" on page 29 describes the instructions associated with the Fixed-Point Processor.

3.2 Fixed-Point Processor Registers

3.2.1 General Purpose Registers

All manipulation of information is done in registers internal to the Fixed-Point Processor. The principal storage internal to the Fixed-Point Processor is a set of 32 general purpose registers (GPRs). See Figure 21.

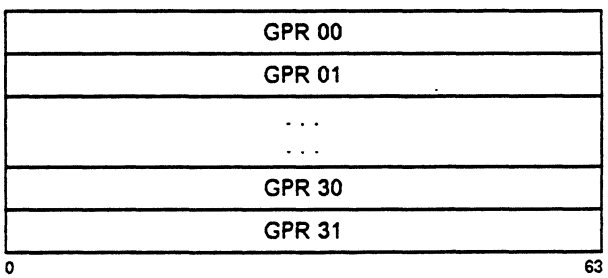


Figure 21. General Purpose Registers

Each GPR is a 64-bit register.

3.2.2 Fixed-Point Exception Register

The Fixed-Point Exception Register (XER) is a 32-bit register.

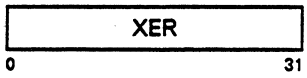


Figure 22. Fixed-Point Exception Register

The bit definitions for the Fixed-Point Exception Register are as shown below. Here $M=0$ in 64-bit mode and $M=32$ in 32-bit mode.

The bits are set based on the operation of an instruction considered as a whole, not on intermediate results (e.g., the *Subtract From Carrying* instruction, the result of which is specified as the sum of three values, sets bits in the Fixed-Point Exception Register based on the entire operation, not on an intermediate sum).

Bit(s) **Description**

0 ***Summary Overflow*** (SO)
The Summary Overflow bit is set to one whenever an instruction sets the Overflow bit to indicate overflow and remains set until it is cleared by an ***mtspr*** instruction (specifying the XER) or an ***mcrxr*** instruction. It is not altered by *Compare* instructions, nor by other instructions (except ***mtspr*** to the XER, and ***mcrxr***) that cannot overflow.

1 ***Overflow*** (OV)
The Overflow bit is set to indicate that an overflow has occurred during execution of an instruction. XO-form *Add* and *Subtract* instructions having OE=1 set it to one if the carry out of bit *M* is not equal to the carry out of bit $M+1$, and set it to zero otherwise. The OV bit is not altered by *Compare* instructions, nor by other instructions (except ***mtspr*** to the XER, and ***mcrxr***) that cannot overflow.

2 ***Carry*** (CA)
In general, the Carry bit is set to indicate that a carry out of bit *M* has occurred during execution of an instruction. *Add Carrying, Subtract From Carrying, Add Extended,* and *Subtract From Extended* instructions set it to one if there is a carry out of bit *M*, and set it to zero otherwise. However, *Shift Right Algebraic* instructions set the CA bit to indicate whether any '1' bits have been shifted out of a negative quantity. The CA bit is not altered by *Compare* instructions, nor by other instructions (except *Shift Right Algebraic*, ***mtspr*** to the XER, and ***mcrxr***) that cannot carry.

3:24 Reserved

25:31 This field specifies the number of bytes to be transferred by a *Load String Indexed* or *Store String Indexed* instruction.

Compatibility Note

For a discussion of Power compatibility with respect to XER bits 16:23, please refer to Appendix G, "Incompatibilities with the Power Architecture" on page 165. For compatibility with future versions of this architecture, these bits should be set to zero.

3.3 Fixed-Point Processor Instructions

This section describes the instructions executed by the Fixed-Point processor.

3.3.1 Storage Access Instructions

The *Storage Access* instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.11.2, "Effective Address Calculation" on page 12.

The order of bytes accessed by halfword, word, and doubleword loads and stores is Big-Endian, unless Little-Endian storage ordering is selected as described in Appendix D, "Little-Endian Byte Ordering" on page 145.

Programming Note

The "la" extended mnemonic permits computing an Effective Address as a *Load* or *Store* instruction would, but loads the address itself into a GPR rather than loading the value that is in storage at that address. This extended mnemonic is described in "Load Address" on page 144.

3.3.1.1 Storage Access Exceptions

Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (*Store* only), or if the program attempts to access storage that is unavailable to it.

When PowerPC is executing with Little-Endian byte ordering, the system alignment error handler will be invoked whenever a load or store instruction is executed that specifies an unaligned operand. See Appendix D, "Little-Endian Byte Ordering" on page 145.

3.3.2 Fixed-Point Load Instructions

The byte, halfword, word, or doubleword in storage addressed by EA is loaded into register RT.

Byte order of PowerPC is Big-Endian by default; see Appendix D, "Little-Endian Byte Ordering" on page 145 for PowerPC systems operated with Little-Endian byte ordering.

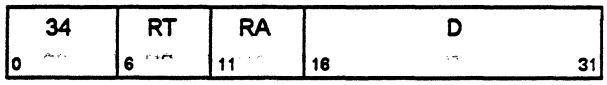
Many of the *Load* instructions have an "update" form, in which register RA is updated with the effective address. For these forms, if RA≠0 and RA≠RT, the effective address is placed into register RA and the storage element (byte, halfword, word, or doubleword) addressed by EA is loaded into RT.

Programming Note

In some implementations, the *Load Algebraic* and *Load with Update* instructions may have greater latency than other types of *Load* instructions. Moreover, *Load with Update* instructions may take longer to execute in some implementations than the corresponding pair of a non-update *Load* instruction and an *Add* instruction.

Load Byte and Zero D-form

lbz RT,D(RA)



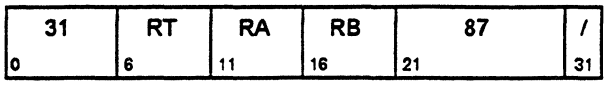

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← 56^0 || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0)+D. The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

Special Registers Altered:
None

Load Byte and Zero Indexed X-form

lbzx RT,RA,RB



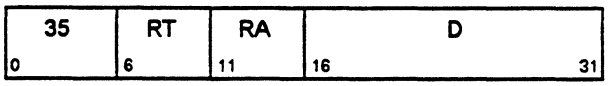

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 56^0 || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

Special Registers Altered:
None

Load Byte and Zero with Update D-form

lbzu RT,D(RA)




```
EA ← (RA) + EXTS(D)
RT ← 56^0 || MEM(EA, 1)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D. The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

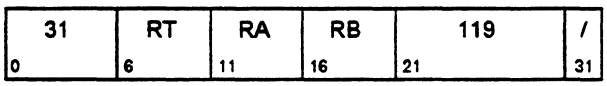
EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None

Load Byte and Zero with Update Indexed X-form

lbzux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← 56^0 || MEM(EA, 1)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

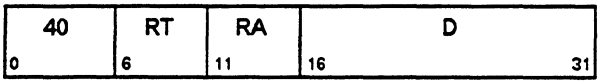
EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None

Load Halfword and Zero D-form

lhz RT,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← 48^0 || MEM(EA, 2)
```

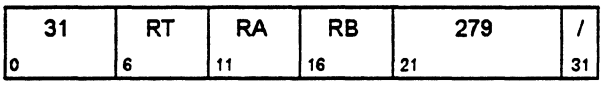
Let the effective address (EA) be the sum (RA|0)+D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:

None

Load Halfword and Zero Indexed X-form

lhzx RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 48^0 || MEM(EA, 2)
```

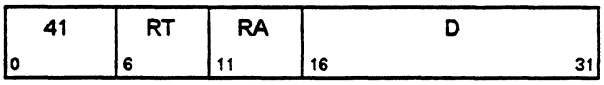
Let the effective address (EA) be the sum (RA|0)+(RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:

None

Load Halfword and Zero with Update D-form

lhzu RT,D(RA)




```
EA ← (RA) + EXTS(D)
RT ← 48^0 || MEM(EA, 2)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

EA is placed into register RA.

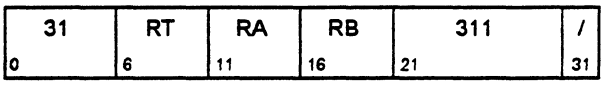
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword and Zero with Update Indexed X-form

lhzux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← 48^0 || MEM(EA, 2)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

EA is placed into register RA.

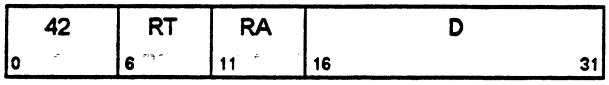
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword Algebraic D-form

lha RT,D(RA)



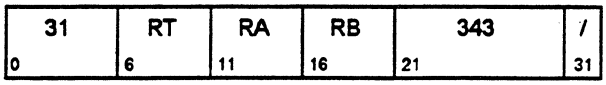

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← EXTS(MEM(EA, 2))
```

Let the effective address (EA) be the sum (RA|0)+D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

Special Registers Altered:
None

Load Halfword Algebraic Indexed X-form

lhax RT,RA,RB



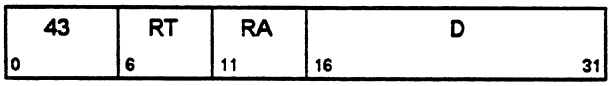

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← EXTS(MEM(EA, 2))
```

Let the effective address (EA) be the sum (RA|0)+(RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

Special Registers Altered:
None

Load Halfword Algebraic with Update D-form

lhau RT,D(RA)




```
EA ← (RA) + EXTS(D)
RT ← EXTS(MEM(EA, 2))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

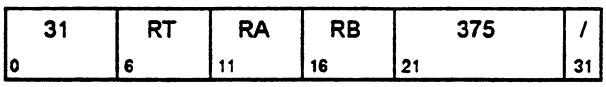
EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None

Load Halfword Algebraic with Update Indexed X-form

lhaux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← EXTS(MEM(EA, 2))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

EA is placed into register RA.

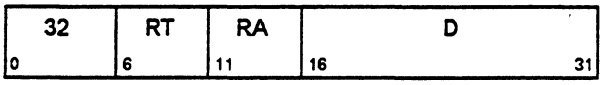
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None

Load Word and Zero D-form

lwz RT,D(RA)

[Power mnemonic: l]




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← 32θ || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RA|0)+D. The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

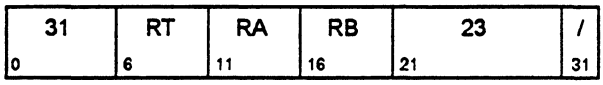
Special Registers Altered:

None

Load Word and Zero Indexed X-form

lwzx RT,RA,RB

[Power mnemonic: lx]




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 32θ || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

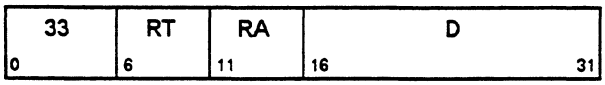
Special Registers Altered:

None

Load Word and Zero with Update D-form

lwzu RT,D(RA)

[Power mnemonic: lu]




```
EA ← (RA) + EXTS(D)
RT ← 32θ || MEM(EA, 4)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D. The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

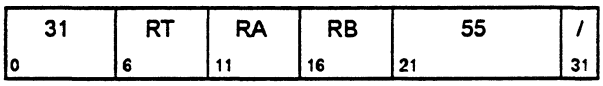
Special Registers Altered:

None

Load Word and Zero with Update Indexed X-form

lwzux RT,RA,RB

[Power mnemonic: lux]




```
EA ← (RA) + (RB)
RT ← 32θ || MEM(EA, 4)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

EA is placed into register RA.

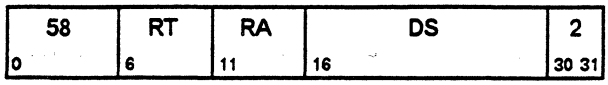
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Word Algebraic DS-form

lwa RT,DS(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(DS||0b00)
RT ← EXTS(MEM(EA, 4))
```

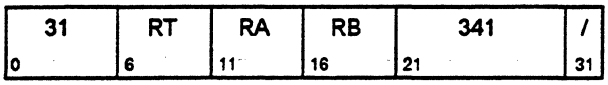
Let the effective address (EA) be the sum (RA|0)+(DS||0b00). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

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.

Special Registers Altered:
None

Load Word Algebraic Indexed X-form

lwax RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← EXTS(MEM(EA, 4))
```

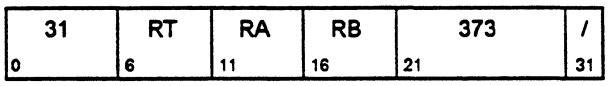
Let the effective address (EA) be the sum (RA|0)+(RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

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.

Special Registers Altered:
None

Load Word Algebraic with Update Indexed X-form

lwaux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← EXTS(MEM(EA, 4))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

EA is placed into register RA.

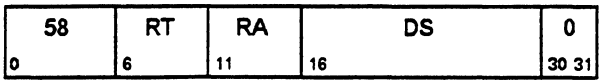
If RA=0 or RA=RT, the instruction form is invalid.

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.

Special Registers Altered:
None

Load Doubleword DS-form

ld RT,DS(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(DS||0b00)
RT ← MEM(EA, 8)
```

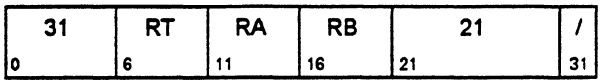
Let the effective address (EA) be the sum (RA|0)+(DS||0b00). The doubleword in storage addressed by EA is loaded into RT.

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.

Special Registers Altered:
None

Load Doubleword Indexed X-form

ldx RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← MEM(EA, 8)
```

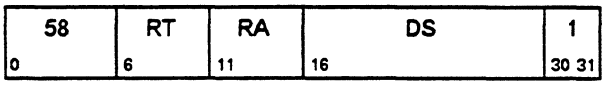
Let the effective address (EA) be the sum (RA|0)+(RB). The doubleword in storage addressed by EA is loaded into RT.

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.

Special Registers Altered:
None

Load Doubleword with Update DS-form

ldu RT,DS(RA)




```
EA ← (RA) + EXTS(DS||0b00)
RT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(DS||0b00). The doubleword in storage addressed by EA is loaded into RT.

EA is placed into register RA.

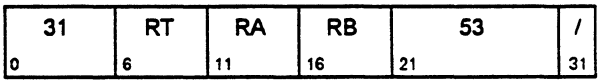
If RA=0 or RA=RT, the instruction form is invalid.

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.

Special Registers Altered:
None

Load Doubleword with Update Indexed X-form

ldux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB). The doubleword in storage addressed by EA is loaded into RT.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

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.

Special Registers Altered:
None

3.3.3 Fixed-Point Store Instructions

The contents of register RS is stored into the byte, halfword, word, or doubleword in storage addressed by EA.

Byte order of PowerPC is Big-Endian by default; see Appendix D, "Little-Endian Byte Ordering" on page 145 for PowerPC systems operated with Little-Endian byte ordering.

Many of the *Store* instructions have an "update" form, in which register RA is updated with the effective address. For these forms, the following rules apply.

- If RA≠0, the effective address is placed into register RA.
- If RS=RA, the contents of register RS is copied to the target storage element and then EA is placed into RA (RS).

Store Byte D-form

stb RS,D(RA)

38	RS	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 1) ← (RS)56:63
```

Let the effective address (EA) be the sum (RA|0)+D. $(RS)_{56:63}$ is stored into the byte in storage addressed by EA.

Special Registers Altered:
None

Store Byte Indexed X-form

stbx RS,RA,RB

31	RS	RA	RB	215	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 1) ← (RS)56:63
```

Let the effective address (EA) be the sum (RA|0)+(RB). $(RS)_{56:63}$ is stored into the byte in storage addressed by EA.

Special Registers Altered:
None

Store Byte with Update D-form

stbu RS,D(RA)

39	RS	RA	D
0	6	11	16 31

```
EA ← (RA) + EXTS(D)
MEM(EA, 1) ← (RS)56:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D. $(RS)_{56:63}$ is stored into the byte in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Byte with Update Indexed X-form

stbux RS,RA,RB

31	RS	RA	RB	247	/
0	6	11	16	21	31

```
EA ← (RA) + (RB)
MEM(EA, 1) ← (RS)56:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB). $(RS)_{56:63}$ is stored into the byte in storage addressed by EA.

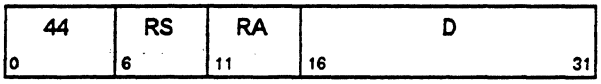
EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Halfword D-form

sth RS,D(RA)



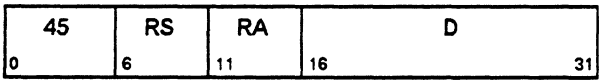

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 2) ← (RS)48:63
```

Let the effective address (EA) be the sum (RA|0)+D. $(RS)_{48:63}$ is stored into the halfword in storage addressed by EA.

Special Registers Altered:
None

Store Halfword with Update D-form

sthu RS,D(RA)




```
EA ← (RA) + EXTS(D)
MEM(EA, 2) ← (RS)48:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D. $(RS)_{48:63}$ is stored into the halfword in storage addressed by EA.

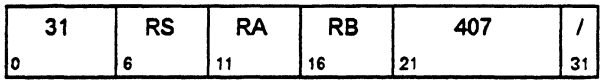
EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Halfword Indexed X-form

sthx RS,RA,RB



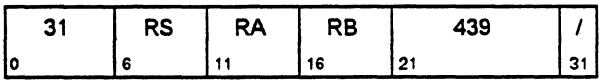

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 2) ← (RS)48:63
```

Let the effective address (EA) be the sum (RA|0)+(RB). $(RS)_{48:63}$ is stored into the halfword in storage addressed by EA.

Special Registers Altered:
None

Store Halfword with Update Indexed X-form

sthux RS,RA,RB




```
EA ← (RA) + (RB)
MEM(EA, 2) ← (RS)48:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB). $(RS)_{48:63}$ is stored into the halfword in storage addressed by EA.

EA is placed into register RA.

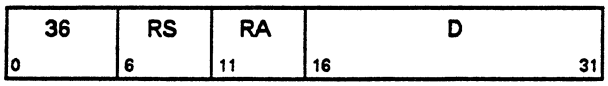
If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Word D-form

stw RS,D(RA)

[Power mnemonic: st]




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 4) ← (RS)32:63
```

Let the effective address (EA) be the sum (RA|0)+D. $(RS)_{32:63}$ is stored into the word in storage addressed by EA.

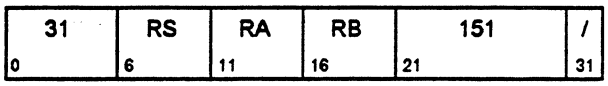
Special Registers Altered:

None

Store Word Indexed X-form

stwx RS,RA,RB

[Power mnemonic: stx]




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (RS)32:63
```

Let the effective address (EA) be the sum (RA|0)+(RB). $(RS)_{32:63}$ is stored into the word in storage addressed by EA.

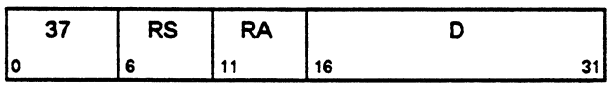
Special Registers Altered:

None

Store Word with Update D-form

stwu RS,D(RA)

[Power mnemonic: stu]




```
EA ← (RA) + EXTS(D)
MEM(EA, 4) ← (RS)32:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D. $(RS)_{32:63}$ is stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

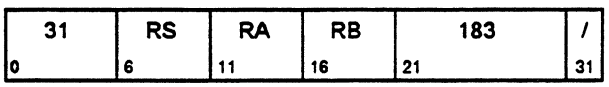
Special Registers Altered:

None

Store Word with Update Indexed X-form

stwux RS,RA,RB

[Power mnemonic: stux]




```
EA ← (RA) + (RB)
MEM(EA, 4) ← (RS)32:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB). $(RS)_{32:63}$ is stored into the word in storage addressed by EA.

EA is placed into register RA.

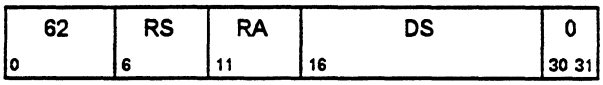
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Doubleword DS-form

std RS,DS(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(DS||0b00)
MEM(EA, 8) ← (RS)
```

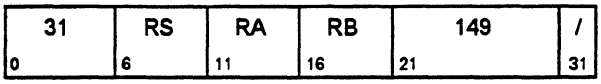
Let the effective address (EA) be the sum (RA|0)+(DS||0b00). (RS) is stored into the doubleword in storage addressed by EA.

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.

Special Registers Altered:
None

Store Doubleword Indexed X-form

stdx RS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (RS)
```

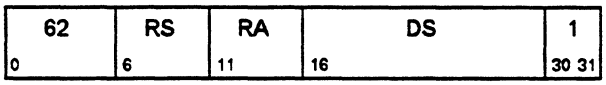
Let the effective address (EA) be the sum (RA|0)+(RB). (RS) is stored into the doubleword in storage addressed by EA.

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.

Special Registers Altered:
None

Store Doubleword with Update DS-form

stdu RS,DS(RA)




```
EA ← (RA) + EXTS(DS||0b00)
MEM(EA, 8) ← (RS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(DS||0b00). (RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

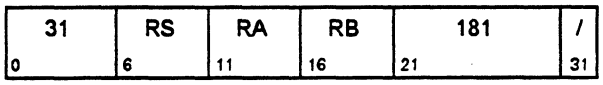
If RA=0, the instruction form is invalid.

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.

Special Registers Altered:
None

Store Doubleword with Update Indexed X-form

stdux RS,RA,RB




```
EA ← (RA) + (RB)
MEM(EA, 8) ← (RS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB). (RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

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.

Special Registers Altered:
None

3.3.4 Fixed-Point Load and Store with Byte Reversal Instructions

When used in a PowerPC system operating with Big-Endian byte order (the default), these instructions have the effect of loading and storing data in Little-Endian order. Likewise, when used in a PowerPC system operating with Little-Endian byte order, these instructions have the effect of loading and storing data in Big-Endian order. See Appendix D, "Little-Endian Byte Ordering" on page 145 for a discussion of byte order.

Programming Note

In some implementations, the *Load Byte-Reverse* instructions may have greater latency than other *Load* instructions.

Load Halfword Byte-Reverse Indexed X-form

lhbrx RT,RA,RB

31	RT	RA	RB	790	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 48 0 || MEM(EA+1, 1) || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0)+(RB). Bits 0:7 of the halfword in storage addressed by EA are loaded into $RT_{56:63}$. Bits 8:15 of the halfword in storage addressed by EA are loaded into $RT_{48:55}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:

None

Load Word Byte-Reverse Indexed X-form

lwbrx RT,RA,RB

[Power mnemonic: lbrx]

31	RT	RA	RB	534	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 32 0 || MEM(EA+3, 1) || MEM(EA+2, 1)
        || MEM(EA+1, 1) || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0)+(RB). Bits 0:7 of the word in storage addressed by EA are loaded into $RT_{56:63}$. Bits 8:15 of the word in storage addressed by EA are loaded into $RT_{48:55}$. Bits 16:23 of the word in storage addressed by EA are loaded into $RT_{40:47}$. Bits 24:31 of the word in storage addressed by EA are loaded into $RT_{32:39}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Store Halfword Byte-Reverse Indexed X-form

sthbrx RS,RA,RB

31	RS	RA	RB	918	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 2) ← (RS)56:63 || (RS)48:55
```

Let the effective address (EA) be the sum (RA|0)+(RB). $(RS)_{56:63}$ are stored into bits 0:7 of the halfword in storage addressed by EA. $(RS)_{48:55}$ are stored into bits 8:15 of the halfword in storage addressed by EA.

Special Registers Altered:

None

Store Word Byte-Reverse Indexed X-form

stwbrx RS,RA,RB

[Power mnemonic: stbrx]

31	RS	RA	RB	662	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (RS)56:63 || (RS)48:55 || (RS)40:47 || (RS)32:39
```

Let the effective address (EA) be the sum (RA|0)+(RB). $(RS)_{56:63}$ are stored into bits 0:7 of the word in storage addressed by EA. $(RS)_{48:55}$ are stored into bits 8:15 of the word in storage addressed by EA. $(RS)_{40:47}$ are stored into bits 16:23 of the word in storage addressed by EA. $(RS)_{32:39}$ are stored into bits 24:31 of the word in storage addressed by EA.

Special Registers Altered:

None

3.3.5 Fixed-Point Load and Store Multiple Instructions

The *Load/Store Multiple* instructions have preferred forms: see Section 1.9.1, "Preferred Instruction Forms" on page 11. In the preferred forms, storage alignment satisfies the following rule.

- The combination of the EA and RT (RS) is such that the low-order byte of GPR 31 is loaded (stored) from (into) the last byte of an aligned quadword in storage.

On PowerPC systems operating with Little-Endian byte order, execution of a *Load Multiple* or *Store Multiple* instruction causes the system alignment trap handler to be invoked. See Appendix D, "Little-Endian Byte Ordering" on page 145.

Compatibility Note

For a discussion of Power compatibility with respect to the alignment of the EA for the *Load Multiple Word* and *Store Multiple Word* instructions, please refer to Appendix G, "Incompatibilities with the Power Architecture" on page 165. For compatibility with future versions of this architecture, these EAs should be word-aligned.

Engineering Note

Causing the system alignment error handler to be invoked if attempt is made to execute a *Load Multiple* or *Store Multiple* instruction having an incorrectly aligned effective address facilitates the debugging of software.

Load Multiple Word D-form

lmw RT,D(RA)

[Power mnemonic: lm]

46	RT	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
r ← RT
do while r ≤ 31
  GPR(r) ← 32₀ || MEM(EA, 4)
  r ← r + 1
  EA ← EA + 4
```

Let n = (32−RT). Let the effective address (EA) be the sum (RA|0)+D.

n consecutive words starting at EA are loaded into the low-order 32 bits of GPRs RT through 31. The high-order 32 bits of these GPRs are set to zero.

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.

If RA is in the range of registers to be loaded or RT=RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Multiple Word D-form

stmw RS,D(RA)

[Power mnemonic: stm]

47	RS	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
r ← RS
do while r ≤ 31
  MEM(EA, 4) ← GPR(r)32:63
  r ← r + 1
  EA ← EA + 4
```

Let n = (32−RS). Let the effective address (EA) be the sum (RA|0)+D.

n consecutive words starting at EA are stored from the low-order 32 bits of GPRs RS through 31.

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.

Special Registers Altered:

None

3.3.6 Fixed-Point Move Assist Instructions

The *Move Assist* instructions allow 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/Store String Indexed instructions of zero length shall have no effect, except that *Load String Indexed* instructions of zero length may set register RT to an undefined value.

The *Load/Store String* instructions have preferred forms: see Section 1.9.1, "Preferred Instruction Forms" on page 11. In the preferred forms, register usage satisfies the following rules.

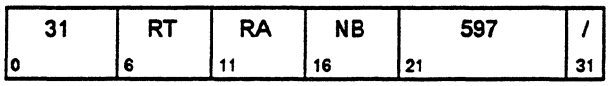
- RS = 5
- RT = 5
- last register loaded/stored $\leq$ 12

On PowerPC systems operating with Little-Endian byte order, execution of a *Load/Store String* instruction causes the system alignment trap handler to be invoked. See Appendix D, "Little-Endian Byte Ordering" on page 145.

Load String Word Immediate X-form

lswi RT,RA,NB

[Power mnemonic: lsi]




```
if RA = 0 then EA ← 0
else           EA ← (RA)
if NB = 0 then n ← 32
else           n ← NB
r ← RT - 1
i ← 32
do while n > 0
  if i = 32 then
    r ← r + 1 (mod 32)
    GPR(r) ← 0
  GPR(r)i:i+7 ← MEM(EA, 1)
  i ← i + 8
  if i = 64 then i ← 32
  EA ← EA + 1
  n ← n - 1
```

Let the effective address (EA) be (RA|0). Let n = NB if NB≠0, n = 32 if NB=0: n is the number of bytes to load. Let nr = CEIL(n÷4): nr is the number of registers to receive data.

n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr−1. Data is loaded into the low-order four bytes of each GPR; the high-order four bytes are set to 0.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If the low-order four bytes of register RT+nr−1 are only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If RA is in the range of registers to be loaded or RT=RA=0, the instruction form is invalid.

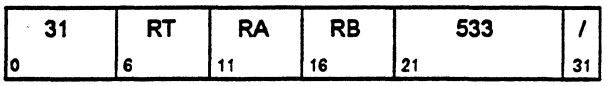
Special Registers Altered:

None

Load String Word Indexed X-form

lswx RT,RA,RB

[Power mnemonic: lsx]




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
n ← XER25:31
r ← RT - 1
i ← 32
RT ← undefined
do while n > 0
  if i = 32 then
    r ← r + 1 (mod 32)
    GPR(r) ← 0
  GPR(r)i:i+7 ← MEM(EA, 1)
  i ← i + 8
  if i = 64 then i ← 32
  EA ← EA + 1
  n ← n - 1
```

Let the effective address (EA) be the sum (RA|0)+(RB). Let n = $XER_{25:31}$: n is the number of bytes to load. Let nr = CEIL(n÷4): nr is the number of registers to receive data.

If n>0, n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr−1. Data is loaded into the low-order four bytes of each GPR; the high-order four bytes are set to 0.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If the low-order four bytes of register RT+nr−1 are only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If n=0, the content of register RT is undefined.

If RA or RB is in the range of registers to be loaded or RT=RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store String Word Immediate X-form

stswi RS,RA,NB

[Power mnemonic: stsi]

31	RS	RA	NB	725	/
0	6	11	16	21	31

```
if RA = 0 then EA ← 0
else           EA ← (RA)
if NB = 0 then n ← 32
else           n ← NB
r ← RS - 1
i ← 32
do while n > 0
  if i = 32 then r ← r + 1 (mod 32)
  MEM(EA, 1) ← GPR(r)i:i+7
  i ← i + 8
  if i = 64 then i ← 32
  EA ← EA + 1
  n ← n - 1
```

Let the effective address (EA) be (RA|0). Let n = NB if NB≠0, n = 32 if NB=0: n is the number of bytes to store. Let nr = CEIL(n÷4): nr is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs RS through RS+nr−1. Data is stored from the low-order four bytes of each GPR.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR 0 if required.

Special Registers Altered:

None

Store String Word Indexed X-form

stswx RS,RA,RB

[Power mnemonic: stsx]

31	RS	RA	RB	661	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
n ← XER25:31
r ← RS - 1
i ← 32
do while n > 0
  if i = 32 then r ← r + 1 (mod 32)
  MEM(EA, 1) ← GPR(r)i:i+7
  i ← i + 8
  if i = 64 then i ← 32
  EA ← EA + 1
  n ← n - 1
```

Let the effective address (EA) be the sum (RA|0)+(RB). Let n = $XER_{25:31}$: n is the number of bytes to store. Let nr = CEIL(n÷4): nr is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs RS through RS+nr−1. Data is stored from the low-order four bytes of each GPR.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR 0 if required.

Special Registers Altered:

None

3.3.7 Storage Synchronization Instructions

The *Storage Synchronization* instructions can be used to control the order in which storage operations are completed with respect to asynchronous events, and the order in which storage operations are seen by other processors and by other mechanisms that access storage. Additional information about these instructions, and about related aspects of storage management, can be found in Book II, *PowerPC Virtual Environment Architecture*, and Book III, *PowerPC Operating Environment Architecture*.

On a PowerPC system operating with Little-Endian byte order the three low-order bits of the Effective Address computed by ***Load Word And Reserve Indexed*** and ***Store Word Conditional Indexed*** are modified before accessing storage. See Appendix D, "Little-Endian Byte Ordering" on page 145.

Architecture Note

The *Load and Reserve* and *Store Conditional* instructions require the EA to be aligned. Software should not attempt to emulate an unaligned *Load and Reserve* or *Store Conditional* instruction, because there is no correct way to define the address associated with the reservation.

Engineering Note

Causing the system alignment error handler to be invoked if attempt is made to execute a *Load and Reserve* or *Store Conditional* instruction having an incorrectly aligned effective address facilitates the debugging of software.

Load Word And Reserve Indexed X-form

lwarx RT,RA,RB

31	RT	RA	RB	20	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
RESERVE ← 1
RESERVE_ADDR ← func(EA)
RT ← 32̄0 || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

This instruction creates a reservation for use by a *Store Word Conditional* instruction. An address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation: the manner in which the address to be associated with the reservation is computed from the EA is described in Book II, *PowerPC Virtual Environment Architecture*.

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.

Special Registers Altered:

None

Load Doubleword And Reserve Indexed X-form

ldarx RT,RA,RB

31	RT	RA	RB	84	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
RESERVE ← 1
RESERVE_ADDR ← func(EA)
RT ← MEM(EA, 8)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The doubleword in storage addressed by EA is loaded into RT.

This instruction creates a reservation for use by a *Store Doubleword Conditional* instruction. An address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation: the manner in which the address to be associated with the reservation is computed from the EA is described in Book II, *PowerPC Virtual Environment Architecture*.

EA must be a multiple of 8. If it is not, the system alignment error handler may be invoked or the results may be boundedly undefined.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

None

Store Word Conditional Indexed X-form

stwcx. RS,RA,RB

31	RS	RA	RB	150	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
if RESERVE then
  MEM(EA, 4) ← (RS)32:63
  RESERVE ← 0
  CR0 ← 0b00 || 0b1 || XERSO
else
  CR0 ← 0b00 || 0b0 || XERSO
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, $(RS)_{32:63}$ is stored into the word in storage addressed by EA and the reservation is cleared.

If a reservation does not exist, the instruction completes without altering storage.

CR Field 0 is set to reflect whether the store operation was performed (i.e., whether a reservation existed when the ***stwcx.*** instruction commenced execution), as follows.

$CR0_{LT\ GT\ EQ\ SO}$ = 0b00 || store_performed || XER_{SO}

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.

Special Registers Altered:

CR0

Store Doubleword Conditional Indexed X-form

stdcx. RS,RA,RB

31	RS	RA	RB	214	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
if RESERVE then
  MEM(EA, 8) ← (RS)
  RESERVE ← 0
  CR0 ← 0b00 || 0b1 || XERSO
else
  CR0 ← 0b00 || 0b0 || XERSO
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, (RS) is stored into the doubleword in storage addressed by EA and the reservation is cleared.

If a reservation does not exist, the instruction completes without altering storage.

CR Field 0 is set to reflect whether the store operation was performed (i.e., whether a reservation existed when the ***stdcx.*** instruction commenced execution), as follows.

$CR0_{LT\ GT\ EQ\ SO}$ = 0b00 || store_performed || XER_{SO}

EA must be a multiple of 8. If it is not, the system alignment error handler may be invoked or the results may be boundedly undefined.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

CR0

Programming Note

The granularity with which reservations are managed is implementation-dependent. Therefore the storage to be accessed by the *Load And Reserve* and *Store Conditional* instructions should be allocated by a system library program. Additional information can be found in Book II, *PowerPC Virtual Environment Architecture*.

Programming Note

When correctly used, 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 doubleword (*Load Doubleword And Reserve* and *Store Doubleword Conditional*) of storage.

One of the requirements for correct use is that *Load Word And Reserve* be paired with *Store Word Conditional*, and *Load Doubleword And Reserve* with *Store Doubleword Conditional*, with the same effective address used for both instructions of the pair. Examples of correct uses of these instructions, to emulate primitives such as "Fetch and Add," "Test and Set," and "Compare and Swap," can be found in Appendix E.1, "Synchronization" on page 153. In general, these instructions should be used only in system programs, which can be invoked by application programs as needed.

At most one reservation exists on any given processor: there are not separate reservations for words and for doublewords.

The address associated with the reservation can be changed by a subsequent *Load And Reserve* instruction.

The conditionality of the *Store Conditional* instruction's store is based only on whether a reservation exists, not on a match between the address associated with the reservation and the address computed from the EA of the *Store Conditional* instruction.

A reservation is cleared if any of the following events occurs.

- the processor having 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 having the reservation, stores to the address associated with the reservation

Synchronize X-form

sync

[Power mnemonic: dcs]

31	///	///	///	598	/
0	6	11	16	21	31

The ***sync*** instruction provides an ordering function for the effects of all instructions executed by a given processor. Executing a ***sync*** instruction ensures that all instructions previously initiated by the given processor appear to have completed before the ***sync*** instruction completes, and that no subsequent instructions are initiated by the given processor until after the ***sync*** instruction completes. When the ***sync*** instruction completes, all storage accesses initiated by the given processor prior to the ***sync*** will have been performed with respect to all other mechanisms that access storage. (See Book II, *PowerPC Virtual Environment Architecture*, for a more complete description. See also the section entitled "Table Update Synchronization Requirements" in Book III, *PowerPC Operating Environment Architecture*, for an exception involving TLB invalidates.)

Special Registers Altered:

None

Programming Note

The ***sync*** instruction can be used to ensure that the results of all stores into a data structure, performed in a "critical section" of a program, are seen by other processors before the data structure is seen as unlocked.

The functions performed by the ***sync*** instruction will normally take a significant amount of time to complete, so indiscriminate use of this instruction may adversely affect performance. In addition, the time required to execute ***sync*** may vary from one execution to another.

The *Enforce In-order Execution of I/O* (***eieio***) instruction, described in Book II, *PowerPC Virtual Environment Architecture*, may be more appropriate than ***sync*** for cases in which the only requirement is to control the order in which storage references are seen by I/O devices.

Engineering Note

Unlike a context synchronizing operation, ***sync*** need not discard prefetched instructions.

3.3.8 Other Fixed-Point Instructions

The remainder of the fixed-point instructions use the content of the General Purpose Registers (GPRs) as source operands, and place results into GPRs, into the fixed-point Exception Register (XER), and into Condition Register fields. In addition, the *Trap* instructions compare the contents of one GPR with a second GPR or immediate data and, if the conditions are met, invoke the system trap handler.

These instructions treat the source operands as signed integers unless the instruction is explicitly identified as performing an unsigned operation.

The X-form and XO-form instructions with Rc=1, and the D-form instruction ***addic.***, ***andi.***, and ***andis.***, set CR Field 0 to characterize the result of the operation. In 64-bit mode, CR Field 0 is set as if the 64-bit result were compared algebraically to zero. In 32-bit mode, this field is set as if the sign-extended low-order 32 bits of the result were compared algebraically to zero.

addic, ***addic.***, ***subfic***, ***addc***, ***subfc***, ***adde***, ***subfe***, ***addme***, ***subfme***, ***addze***, and ***subfze*** always set CA, to reflect the carry out of bit 0 in 64-bit mode and out of bit 32 in 32-bit mode. The XO-forms set SO and OV when OE=1, to reflect overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode.

Unless otherwise noted and when appropriate, when CR Field 0 and the XER are set they reflect the value placed in the target register.

Programming Note

Instructions with the OE bit set or which set CA may execute slowly or may prevent the execution of subsequent instructions until the operation is completed.

3.3.9 Fixed-Point Arithmetic Instructions

Extended mnemonics for addition and subtraction

Several extended mnemonics are provided that use the *Add Immediate* and *Add Immediate Shifted* instructions to load an immediate value or an address into a target register. Some of these are shown as examples with the two instructions.

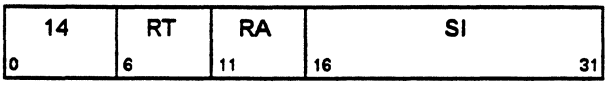
The PowerPC Architecture supplies *Subtract From* instructions, which subtract the second operand from the third. A set of extended mnemonics is provided that use the more "normal" order, in which the third operand is subtracted from the second, with the third operand being either an immediate field or a register. Some of these are shown as examples with the appropriate *Add* and *Subtract From* instructions.

See Appendix C, "Assembler Extended Mnemonics" on page 133 for additional extended mnemonics.

Add Immediate D-form

addi RT,RA,SI

[Power mnemonic: cal]




```
if RA = 0 then RT ← EXTS(SI)
else          RT ← (RA) + EXTS(SI)
```

The sum (RA|0) + SI is placed into register RT.

Special Registers Altered:

None

Extended Mnemonics:

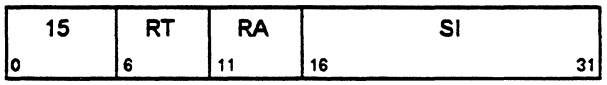
Examples of extended mnemonics for *Add Immediate*:

Extended:		*Equivalent to:*	
li	Rx,value	addi	Rx,0,value
la	Rx,disp(Ry)	addi	Rx,Ry,disp
subi	Rx,Ry,value	addi	Rx,Ry,—value

Add Immediate Shifted D-form

addis RT,RA,SI

[Power mnemonic: cau]




```
if RA = 0 then RT ← EXTS(SI || 16 0)
else          RT ← (RA) + EXTS(SI || 16 0)
```

The sum (RA|0) + (SI || 0x0000) is placed into register RT.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Add Immediate Shifted*:

Extended:		*Equivalent to:*	
lis	Rx,value	addis	Rx,0,value
subis	Rx,Ry,value	addis	Rx,Ry,—value

Add XO-form

add	RT,RA,RB	(OE=0 Rc=0)
add.	RT,RA,RB	(OE=0 Rc=1)
addo	RT,RA,RB	(OE=1 Rc=0)
addo.	RT,RA,RB	(OE=1 Rc=1)

[Power mnemonics: cax, cax., caxo, caxo.]

31	RT	RA	RB	OE	266	Rc
0	6	11	16	21	22	31

RT ← (RA) + (RB)

The sum (RA) + (RB) is placed into register RT.

Special Registers Altered:

CR0 (if Rc=1)
SO OV (if OE=1)

Subtract From XO-form

subf	RT,RA,RB	(OE=0 Rc=0)
subf.	RT,RA,RB	(OE=0 Rc=1)
subfo	RT,RA,RB	(OE=1 Rc=0)
subfo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	40	Rc
0	6	11	16	21	22	31

RT ← ¬(RA) + (RB) + 1

The sum ¬(RA) + (RB) +1 is placed into register RT.

Special Registers Altered:

CR0 (if Rc=1)
SO OV (if OE=1)

Extended Mnemonics:

Example of extended mnemonics for *Subtract From*:

Extended:		*Equivalent to:*	
sub	Rx,Ry,Rz	subf	Rx,Rz,Ry

Programming Note

addi, addis, add, and ***subf*** are the preferred instructions for addition and subtraction, because they set few status bits.

Notice that ***addi*** and ***addis*** use the value 0, not the contents of GPR 0, if RA=0.

Add Immediate Carrying D-form

addic RT,RA,SI

[Power mnemonic: ai]

12	RT	RA	SI
0	6	11	16 31

RT ← (RA) + EXTS(SI)

The sum (RA) + SI is placed into register RT.

Special Registers Altered:

CA

Extended Mnemonics:

Example of extended mnemonics for *Add Immediate Carrying*:

Extended:		*Equivalent to:*	
subic	Rx,Ry,value	addic	Rx,Ry,–value

Add Immediate Carrying and Record D-form

addic. RT,RA,SI

[Power mnemonic: ai.]

13	RT	RA	SI
0	6	11	16 31

RT ← (RA) + EXTS(SI)

The sum (RA) + SI is placed into register RT.

Special Registers Altered:

CR0 CA

Extended Mnemonics:

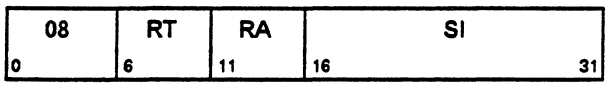
Example of extended mnemonics for *Add Immediate Carrying and Record*:

Extended:		*Equivalent to:*	
subic.	Rx,Ry,value	addic.	Rx,Ry,–value

Subtract From Immediate Carrying D-form

subfic RT,RA,SI

[Power mnemonic: sfi]



RT ← ¬(RA) + EXTS(SI) + 1

The sum ¬(RA) + SI + 1 is placed into register RT.

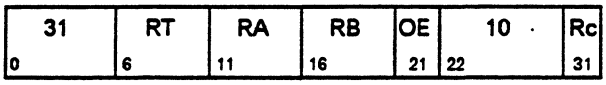
Special Registers Altered:

CA

Add Carrying XO-form

addc	RT,RA,RB	(OE=0 Rc=0)
addc.	RT,RA,RB	(OE=0 Rc=1)
addco	RT,RA,RB	(OE=1 Rc=0)
addco.	RT,RA,RB	(OE=1 Rc=1)

[Power mnemonics: a, a., ao, ao.]



RT ← (RA) + (RB)

The sum (RA) + (RB) is placed into register RT.

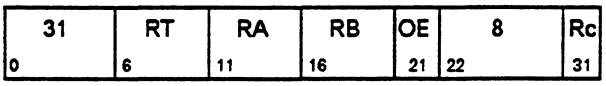
Special Registers Altered:

CA	
CR0	(if Rc=1)
SO OV	(if OE=1)

Subtract From Carrying XO-form

subfc	RT,RA,RB	(OE=0 Rc=0)
subfc.	RT,RA,RB	(OE=0 Rc=1)
subfco	RT,RA,RB	(OE=1 Rc=0)
subfco.	RT,RA,RB	(OE=1 Rc=1)

[Power mnemonics: sf, sf., sfo, sfo.]



RT ← ¬(RA) + (RB) + 1

The sum ¬(RA) + (RB) + 1 is placed into register RT.

Special Registers Altered:

CA	
CR0	(if Rc=1)
SO OV	(if OE=1)

Extended Mnemonics:

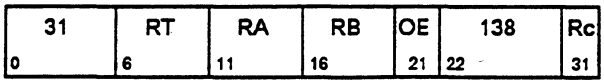
Example of extended mnemonics for *Subtract From Carrying*:

Extended:	*Equivalent to:*
subc Rx,Ry,Rz	subfc Rx,Rz,Ry

Add Extended XO-form

adde	RT,RA,RB	(OE=0 Rc=0)
adde.	RT,RA,RB	(OE=0 Rc=1)
addeo	RT,RA,RB	(OE=1 Rc=0)
addeo.	RT,RA,RB	(OE=1 Rc=1)

[Power mnemonics: ae, ae., aeo, aeo.]



RT ← (RA) + (RB) + CA

The sum (RA) + (RB) + CA is placed into register RT.

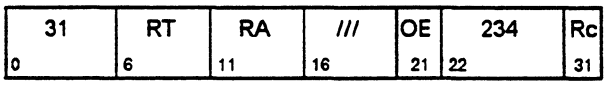
Special Registers Altered:

CA
CR0 (if Rc=1)
SO OV (if OE=1)

Add To Minus One Extended XO-form

addme	RT,RA	(OE=0 Rc=0)
addme.	RT,RA	(OE=0 Rc=1)
addmeo	RT,RA	(OE=1 Rc=0)
addmeo.	RT,RA	(OE=1 Rc=1)

[Power mnemonics: ame, ame., ameo, ameo.]



RT ← (RA) + CA - 1

The sum (RA) + CA + $^{64}1$ is placed into register RT.

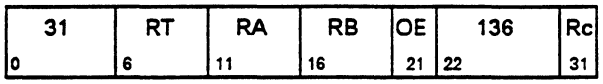
Special Registers Altered:

CA
CR0 (if Rc=1)
SO OV (if OE=1)

Subtract From Extended XO-form

subfe	RT,RA,RB	(OE=0 Rc=0)
subfe.	RT,RA,RB	(OE=0 Rc=1)
subfeo	RT,RA,RB	(OE=1 Rc=0)
subfeo.	RT,RA,RB	(OE=1 Rc=1)

[Power mnemonics: sfe, sfe., sfeo, sfeo.]



RT ← ¬(RA) + (RB) + CA

The sum ¬(RA) + (RB) + CA is placed into register RT.

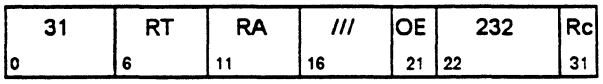
Special Registers Altered:

CA
CR0 (if Rc=1)
SO OV (if OE=1)

Subtract From Minus One Extended XO-form

subfme	RT,RA	(OE=0 Rc=0)
subfme.	RT,RA	(OE=0 Rc=1)
subfmeo	RT,RA	(OE=1 Rc=0)
subfmeo.	RT,RA	(OE=1 Rc=1)

[Power mnemonics: sfme, sfme., sfmeo, sfmeo.]



RT ← ¬(RA) + CA - 1

The sum ¬(RA) + CA + $^{64}1$ is placed into register RT.

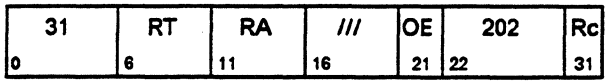
Special Registers Altered:

CA
CR0 (if Rc=1)
SO OV (if OE=1)

Add To Zero Extended XO-form

addze	RT,RA	(OE=0 Rc=0)
addze.	RT,RA	(OE=0 Rc=1)
addzeo	RT,RA	(OE=1 Rc=0)
addzeo.	RT,RA	(OE=1 Rc=1)

[Power mnemonics: aze, aze., azeo, azeo.]



RT ← (RA) + CA

The sum (RA) + CA is placed into register RT.

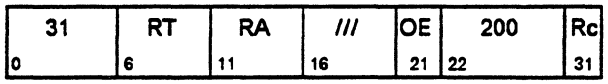
Special Registers Altered:

CA
CR0 (if Rc=1)
SO OV (if OE=1)

Subtract From Zero Extended XO-form

subfze	RT,RA	(OE=0 Rc=0)
subfze.	RT,RA	(OE=0 Rc=1)
subfzeo	RT,RA	(OE=1 Rc=0)
subfzeo.	RT,RA	(OE=1 Rc=1)

[Power mnemonics: sfze, sfze., sfzeo, sfzeo.]



RT ← ¬(RA) + CA

The sum ¬(RA) + CA is placed into register RT.

Special Registers Altered:

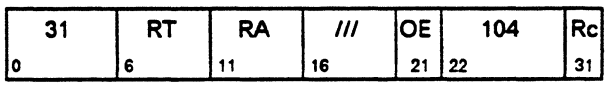
CA
CR0 (if Rc=1)
SO OV (if OE=1)

Programming Note

The setting of CA by the *Add* and *Subtract* instructions, including the Extended versions thereof, is mode-dependent. If a sequence of these instructions is used to perform extended-precision addition or subtraction, the same mode should be used throughout the sequence.

Negate XO-form

neg	RT,RA	(OE=0 Rc=0)
neg.	RT,RA	(OE=0 Rc=1)
nego	RT,RA	(OE=1 Rc=0)
nego.	RT,RA	(OE=1 Rc=1)



RT ← ¬(RA) + 1

The sum ¬(RA) + 1 is placed into register RT.

If executing in 64-bit mode and register RA contains the most negative 64-bit number (0x8000_0000_0000_0000), the result is the most negative number and, if OE=1, OV is set. Similarly, if executing in 32-bit mode and $(RA)_{32:63}$ contains the most negative 32-bit number (0x8000_0000), the low-order 32 bits of the result contain the most negative 32-bit number and, if OE=1, OV is set.

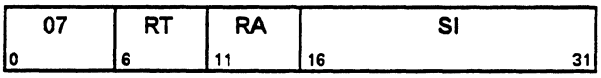
Special Registers Altered:

CR0 (if Rc=1)
SO OV (if OE=1)

Multiply Low Immediate D-form

mulli RT,RA,SI

[Power mnemonic: muli]



$prod_{0:79} \leftarrow (RA) \times SI$
$RT \leftarrow prod_{16:79}$

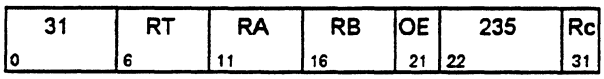
The 64-bit first multiplicand is (RA). The 16-bit second multiplicand is SI. The low-order 64 bits of the 80-bit product of the multiplicands are placed into register RT.

Special Registers Altered:
None

Multiply Low Word XO-form

mullw	RT,RA,RB	(OE=0 Rc=0)
mullw.	RT,RA,RB	(OE=0 Rc=1)
mullwo	RT,RA,RB	(OE=1 Rc=0)
mullwo.	RT,RA,RB	(OE=1 Rc=1)

[Power mnemonics: muls, muls., mulso, mulso.]



$RT \leftarrow (RA)_{32:63} \times (RB)_{32:63}$

The 32-bit operands are the low-order 32 bits of RA and RB. The 64-bit product of the operands is placed into register RT.

If OE=1, then SO and OV are set to one if the product cannot be represented in 32 bits.

Both the operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0	(if Rc=1)
SO OV	(if OE=1)

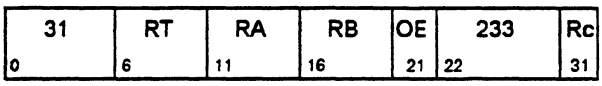
Programming Notes

For ***mulli*** and ***mullw***, the low-order 32 bits of the product are the correct 32-bit product for 32-bit mode.

The XO-form *multiply* instructions may execute faster on some implementations if RB contains the operand having the smaller absolute value.

Multiply Low Doubleword XO-form

mulld	RT,RA,RB	(OE=0 Rc=0)
mulld.	RT,RA,RB	(OE=0 Rc=1)
mulldo	RT,RA,RB	(OE=1 Rc=0)
mulldo.	RT,RA,RB	(OE=1 Rc=1)



$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{64:127}$

The 64-bit operands are (RA) and (RB). The low-order 64 bits of the 128-bit product of the operands are placed into register RT.

If OE=1, then SO and OV are set to one if the product cannot be represented in 64 bits.

Both the operands and the product are interpreted as signed integers. (However, the result in RT is independent of whether the operands are interpreted as signed or unsigned integers.)

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.

Special Registers Altered:

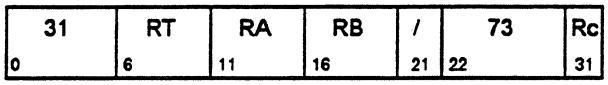
CR0	(if Rc=1)
SO OV	(if OE=1)

Editor's Note

It is proposed to replace the ***mull*** instruction by two: ***mullw*** and ***mulld***. This change has not been officially adopted by the PAWG. However, it is included here for early dissemination.

Multiply High Doubleword XO-form

mulhd RT,RA,RB (Rc=0)
mulhd. RT,RA,RB (Rc=1)



$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{0:63}$

The 64-bit multiplicands are (RA) and (RB). The high-order 64 bits of the 128-bit product of the multiplicands are placed into register RT.

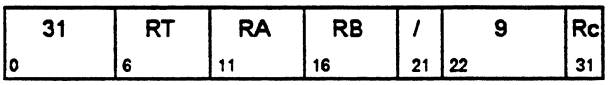
Both the multiplicands and the product are interpreted as signed integers.

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.

Special Registers Altered:
CR0 (if Rc=1)

Multiply High Doubleword Unsigned XO-form

mulhdu RT,RA,RB (Rc=0)
mulhdu. RT,RA,RB (Rc=1)



$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{0:63}$

The 64-bit multiplicands are (RA) and (RB). The high-order 64 bits of the 128-bit product of the multiplicands are placed into register RT.

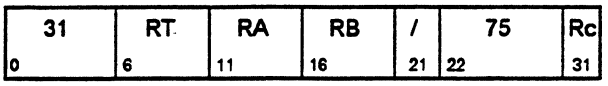
Both the multiplicands and the product are interpreted as unsigned integers.

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.

Special Registers Altered:
CR0 (if Rc=1)

Multiply High Word XO-form

mulhw RT,RA,RB (Rc=0)
mulhw. RT,RA,RB (Rc=1)



$prod_{0:63} \leftarrow (RA)_{32:63} \times (RB)_{32:63}$
$RT_{32:63} \leftarrow prod_{0:31}$
$RT_{0:31} \leftarrow$ undefined

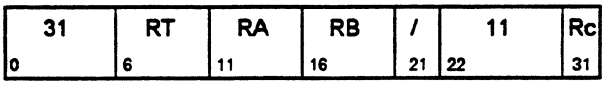
The 32-bit multiplicands are the low-order 32 bits of RA and of RB. The high-order 32 bits of the 64-bit product of the multiplicands are placed into $RT_{32:63}$. $(RT)_{0:31}$ are undefined.

Both the multiplicands and the product are interpreted as signed integers.

Special Registers Altered:
CR0 (if Rc=1)

Multiply High Word Unsigned XO-form

mulhwu RT,RA,RB (Rc=0)
mulhwu. RT,RA,RB (Rc=1)



$prod_{0:63} \leftarrow (RA)_{32:63} \times (RB)_{32:63}$
$RT_{32:63} \leftarrow prod_{0:31}$
$RT_{0:31} \leftarrow$ undefined

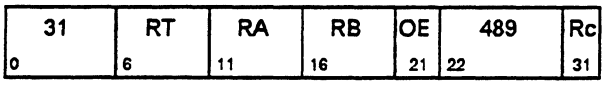
The 32-bit multiplicands are the low-order 32 bits of RA and of RB. The high-order 32 bits of the 64-bit product of the multiplicands are placed into $RT_{32:63}$. $(RT)_{0:31}$ are undefined.

Both the multiplicands and the product are interpreted as unsigned integers.

Special Registers Altered:
CR0 (if Rc=1)

Divide Doubleword XO-form

divd	RT,RA,RB	(OE=0 Rc=0)
divd.	RT,RA,RB	(OE=0 Rc=1)
divdo	RT,RA,RB	(OE=1 Rc=0)
divdo.	RT,RA,RB	(OE=1 Rc=1)



$dividend_{0:63} \leftarrow (RA)$
$divisor_{0:63} \leftarrow (RB)$
$RT \leftarrow dividend \div divisor$

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit quotient of the dividend and divisor is placed into RT. The remainder is not supplied as a result.

Both the dividend and the divisor are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r < |divisor|$ if the dividend is nonnegative, and $-|divisor| < r \le 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

```
0x8000_0000_0000_0000 ÷ -1
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV is set to 1.

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.

Special Registers Altered:

CR0	(if Rc=1)
SO OV	(if OE=1)

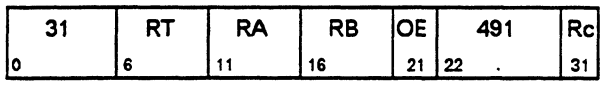
Programming Note

The 64-bit signed remainder of dividing (RA) by (RB) can be computed as follows, except in the case that (RA) = -2^{63} and (RB) = −1.

```
divd   RT,RA,RB    # RT = quotient
mulld  RT,RT,RB    # RT = quotient*divisor
subf   RT,RT,RA    # RT = remainder
```

Divide Word XO-form

divw	RT,RA,RB	(OE=0 Rc=0)
divw.	RT,RA,RB	(OE=0 Rc=1)
divwo	RT,RA,RB	(OE=1 Rc=0)
divwo.	RT,RA,RB	(OE=1 Rc=1)



$dividend_{0:63} \leftarrow EXTS((RA)_{32:63})$
$divisor_{0:63} \leftarrow EXTS((RB)_{32:63})$
$RT_{32:63} \leftarrow dividend \div divisor$
$RT_{0:31} \leftarrow$ undefined

The 64-bit dividend is the sign-extended value of $(RA)_{32:63}$. The 64-bit divisor is the sign-extended value of $(RB)_{32:63}$. The 64-bit quotient is formed. The low-order 32 bits of the 64-bit quotient are placed into $RT_{32:63}$. $(RT)_{0:31}$ are undefined. The remainder is not supplied as a result.

Both the dividend and the divisor are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r < |divisor|$ if the dividend is nonnegative, and $-|divisor| < r \le 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

```
0x8000_0000 ÷ -1
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV is set to 1.

Special Registers Altered:

CR0	(if Rc=1)
SO OV	(if OE=1)

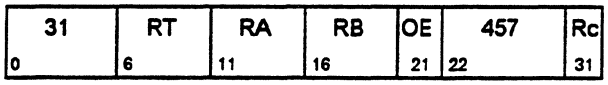
Programming Note

The 32-bit signed remainder of dividing $(RA)_{32:63}$ by $(RB)_{32:63}$ can be computed as follows, except in the case that (RA) = -2^{31} and (RB) = −1.

```
divw   RT,RA,RB    # RT = quotient
mullw  RT,RT,RB    # RT = quotient*divisor
subf   RT,RT,RA    # RT = remainder
```

Divide Doubleword Unsigned XO-form

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)



$dividend_{0:63} \leftarrow (RA)$
$divisor_{0:63} \leftarrow (RB)$
RT ← dividend ÷ divisor

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit quotient of the dividend and divisor is placed into RT. The remainder is not supplied as a result.

Both the dividend and the divisor are interpreted as unsigned integers. The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \leq r < divisor$.

If an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In this case, if OE=1 then OV is set to 1.

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.

Special Registers Altered:

CR0	(if Rc=1)
SO OV	(if OE=1)

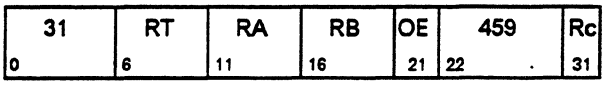
Programming Note

The 64-bit unsigned remainder of dividing (RA) by (RB) can be computed as follows.

```
divdu  RT,RA,RB    # RT = quotient
mulld  RT,RT,RB    # RT = quotient*divisor
subf   RT,RT,RA    # RT = remainder
```

Divide Word Unsigned XO-form

divwu	RT,RA,RB	(OE=0 Rc=0)
divwu.	RT,RA,RB	(OE=0 Rc=1)
divwuo	RT,RA,RB	(OE=1 Rc=0)
divwuo.	RT,RA,RB	(OE=1 Rc=1)



$dividend_{0:63} \leftarrow {}^{32}0 \parallel (RA)_{32:63}$
$divisor_{0:63} \leftarrow {}^{32}0 \parallel (RB)_{32:63}$
$RT_{32:63} \leftarrow$ dividend ÷ divisor
$RT_{0:31} \leftarrow$ undefined

The 64-bit dividend is the zero-extended value of $(RA)_{32:63}$. The 64-bit divisor is the zero-extended value of $(RB)_{32:63}$. The 64-bit quotient is formed. The low-order 32 bits of the 64-bit quotient are placed into $RT_{32:63}$. $(RT)_{0:31}$ are undefined. The remainder is not supplied as a result.

Both the dividend and the divisor are interpreted as unsigned integers. The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \leq r < divisor$.

If an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In this case, if OE=1 then OV is set to 1.

Special Registers Altered:

CR0	(if Rc=1)
SO OV	(if OE=1)

Programming Note

The 32-bit unsigned remainder of dividing $(RA)_{32:63}$ by $(RB)_{32:63}$ can be computed as follows.

```
divwu  RT,RA,RB    # RT = quotient
mullw  RT,RT,RB    # RT = quotient*divisor
subf   RT,RT,RA    # RT = remainder
```

3.3.10 Fixed-Point Compare Instructions

The Fixed-Point *Compare* instructions algebraically or logically compare the contents of register RA with (1) the sign-extended value of the SI field, (2) the UI field, or (3) the contents of register RB. Algebraic comparison compares two signed integers. Logical comparison compares two unsigned integers.

For 64-bit implementations, the L field controls whether the operands are treated as 64- or 32-bit quantities, as follows:

L	Operand length
0	32-bit operands
1	64-bit operands

When the operands are treated as 32-bit signed quantities, bit 32 of the register (RA or RB) is the sign bit.

For 32-bit implementations, the L field must be zero.

The *Compare* instructions set one bit in the leftmost three bits of the designated CR field to one, and the other two to zero. XER_{SO} is copied into bit 3 of the designated CR field.

The CR field is set as follows.

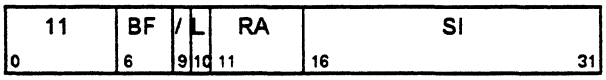
Bit	Name	Description
0	LT	(RA) < SI, UI, or (RB)
1	GT	(RA) > SI, UI, or (RB)
2	EQ	(RA) = SI, UI, or (RB)
3	SO	Summary Overflow from the XER

Extended mnemonics for compares

A set of extended mnemonics is provided so that compares can be coded with the operand length as part of the instruction mnemonic rather than as a numeric operand. Some of these are shown as examples with the *Compare* instructions. The extended mnemonics for doubleword comparisons are available only in 64-bit implementations. See Appendix C, "Assembler Extended Mnemonics" on page 133 for additional extended mnemonics.

Compare Immediate D-form

cmpi BF,L,RA,SI




```
if L = 0 then a ← EXTS((RA)32:63)
         else a ← (RA)
if       a < EXTS(SI) then c ← 0b100
else if a > EXTS(SI) then c ← 0b010
else                       c ← 0b001
CR4×BF:4×BF+3 ← c || XERSO
```

The contents of register RA ($(RA)_{32:63}$ sign-extended to 64 bits if L=0) is compared with the sign-extended value of the SI field, treating the operands as signed integers. The result of the comparison is placed into CR field BF.

In 32-bit implementations, if L=1 the instruction form is invalid.

Special Registers Altered:
CR field BF

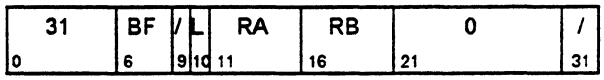
Extended Mnemonics:

Examples of extended mnemonics for *Compare Immediate*:

Extended:		*Equivalent to:*	
cmpdi	Rx,value	cmpi	0,1,Rx,value
cmpwi	cr3,Rx,value	cmpi	3,0,Rx,value

Compare X-form

cmp BF,L,RA,RB




```
if L = 0 then a ← EXTS((RA)32:63)
              b ← EXTS((RB)32:63)
         else a ← (RA)
              b ← (RB)
if       a < b then c ← 0b100
else if a > b then c ← 0b010
else               c ← 0b001
CR4×BF:4×BF+3 ← c || XERSO
```

The contents of register RA ($(RA)_{32:63}$ if L=0) is compared with the contents of register RB ($(RB)_{32:63}$ if L=0), treating the operands as signed integers. The result of the comparison is placed into CR field BF.

In 32-bit implementations, if L=1 the instruction form is invalid.

Special Registers Altered:
CR field BF

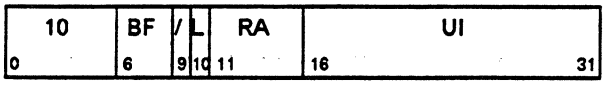
Extended Mnemonics:

Examples of extended mnemonics for *Compare*:

Extended:		*Equivalent to:*	
cmpd	Rx,Ry	cmp	0,1,Rx,Ry
cmpw	cr3,Rx,Ry	cmp	3,0,Rx,Ry

Compare Logical Immediate D-form

cmpli BF,L,RA,UI




```
if L = 0 then a ← 32ℓ0 || (RA)32:63
         else a ← (RA)
if      a ≺ (48ℓ0 || UI) then c ← ℓb100
else if a ≻ (48ℓ0 || UI) then c ← ℓb010
else                           c ← ℓb001
CR4×BF:4×BF+3 ← c || XERSO
```

The contents of register RA ($(RA)_{32:63}$ zero-extended to 64 bits if L=0) is compared with $^{48}0$ || UI, treating the operands as unsigned integers. The result of the comparison is placed into CR field BF.

In 32-bit implementations, if L=1 the instruction form is invalid.

Special Registers Altered:

CR field BF

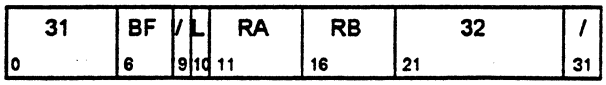
Extended Mnemonics:

Examples of extended mnemonics for *Compare Logical Immediate*:

Extended:	*Equivalent to:*
cmpldi Rx,value	cmpli 0,1,Rx,value
cmplwi cr3,Rx,value	cmpli 3,0,Rx,value

Compare Logical X-form

cmpl BF,L,RA,RB




```
if L = 0 then a ← 32ℓ0 || (RA)32:63
              b ← 32ℓ0 || (RB)32:63
         else a ← (RA)
              b ← (RB)
if      a ≺ b then c ← ℓb100
else if a ≻ b then c ← ℓb010
else               c ← ℓb001
CR4×BF:4×BF+3 ← c || XERSO
```

The contents of register RA ($(RA)_{32:63}$ if L=0) is compared with the contents of register RB ($(RB)_{32:63}$ if L=0), treating the operands as unsigned integers. The result of the comparison is placed into CR field BF.

In 32-bit implementations, if L=1 the instruction form is invalid.

Special Registers Altered:

CR field BF

Extended Mnemonics:

Examples of extended mnemonics for *Compare Logical*:

Extended:	*Equivalent to:*
cmpld Rx,Ry	cmpl 0,1,Rx,Ry
cmplw cr3,Rx,Ry	cmpl 3,0,Rx,Ry

3.3.11 Fixed-Point Trap Instructions

The *Trap* instructions are provided to test for a specified set of conditions. If any of the conditions tested by a *Trap* instruction are met, the system trap handler is invoked. If the tested conditions are not met, instruction execution continues normally.

The contents of register RA is compared with either the sign-extended SI field or with the contents of register RB depending on the *Trap* instruction. For ***tdi*** and ***td***, the entire contents of RA (and RB) participate in the comparison; for ***twi*** and ***tw***, only the contents of the low-order 32 bits of RA (and RB) participate in the comparison.

This comparison results in five conditions which are ANDed with TO. If the result is not 0 the system trap handler is invoked. These conditions are:

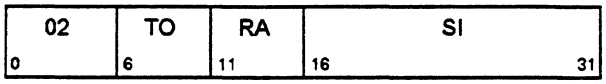
TO bit	ANDed with Condition
0	Less Than
1	Greater Than
2	Equal
3	Logically Less Than
4	Logically Greater Than

Extended mnemonics for traps

A set of extended mnemonics is provided so that traps can be coded with the condition as part of the instruction mnemonic rather than as a numeric operand. Some of these are shown as examples with the *Trap* instructions. See Appendix C, "Assembler Extended Mnemonics" on page 133 for additional extended mnemonics.

Trap Doubleword Immediate D-form

tdi TO,RA,SI




```
a ← (RA)
if (a < EXTS(SI)) & TO0 then TRAP
if (a > EXTS(SI)) & TO1 then TRAP
if (a = EXTS(SI)) & TO2 then TRAP
if (a ≮ EXTS(SI)) & TO3 then TRAP
if (a ≯ EXTS(SI)) & TO4 then TRAP
```

The contents of register RA is compared with the sign-extended SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

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.

Special Registers Altered:
None

Extended Mnemonics:

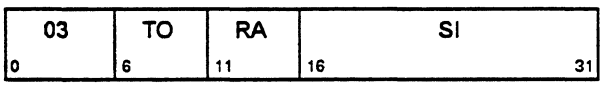
Examples of extended mnemonics for *Trap Doubleword Immediate*:

Extended:		*Equivalent to:*	
tdlti	Rx,value	tdi	16,Rx,value
tdnei	Rx,value	tdi	24,Rx,value

Trap Word Immediate D-form

twi TO,RA,SI

[Power mnemonic: ti]




```
a ← EXTS((RA)32:63)
if (a < EXTS(SI)) & TO0 then TRAP
if (a > EXTS(SI)) & TO1 then TRAP
if (a = EXTS(SI)) & TO2 then TRAP
if (a ≮ EXTS(SI)) & TO3 then TRAP
if (a ≯ EXTS(SI)) & TO4 then TRAP
```

The contents of $RA_{32:63}$ is compared with the sign-extended SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Special Registers Altered:
None

Extended Mnemonics:

Examples of extended mnemonics for *Trap Word Immediate*:

Extended:		*Equivalent to:*	
twgti	Rx,value	twi	8,Rx,value
twllei	Rx,value	twi	6,Rx,value

Trap Doubleword X-form

td TO,RA,RB

31	TO	RA	RB	68	/
0	6	11	16	21	31

```
a ← (RA)
b ← (RB)
if (a < b) & TO0 then TRAP
if (a > b) & TO1 then TRAP
if (a = b) & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4 then TRAP
```

The contents of register RA is compared with the contents of register RB. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

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.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Trap Doubleword*:

Extended:		*Equivalent to:*	
tdge	Rx,Ry	td	12,Rx,Ry
tdlnl	Rx,Ry	td	5,Rx,Ry

Trap Word X-form

tw TO,RA,RB

[Power mnemonic: t]

31	TO	RA	RB	4	/
0	6	11	16	21	31

```
a ← EXTS((RA)32:63)
b ← EXTS((RB)32:63)
if (a < b) & TO0 then TRAP
if (a > b) & TO1 then TRAP
if (a = b) & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4 then TRAP
```

The contents of $RA_{32:63}$ is compared with the contents of $RB_{32:63}$. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Trap Word*:

Extended:		*Equivalent to:*	
tweq	Rx,Ry	tw	4,Rx,Ry
twlge	Rx,Ry	tw	5,Rx,Ry
trap		tw	31,0,0

3.3.12 Fixed-Point Logical Instructions

The *Logical* instructions perform bit-parallel operations on 64-bit operands.

The X-form *Logical* instructions with Rc=1, and the D-form *Logical* instructions ***andi.*** and ***andis.***, set CR Field 0 to characterize the result of the logical operation. In 64-bit mode, CR Field 0 is set as if the 64-bit result were algebraically compared to zero. In 32-bit mode, these fields are set as if the sign-extended low-order 32 bits of the result were algebraically compared to zero. The X-form *Logical* instructions with Rc=0, and the remaining D-form *Logical* instructions, do not change the Condition Register. The *Logical* instructions do not change the SO, OV, and CA bits in the XER.

Extended mnemonics for logical operations

An extended mnemonic is provided that generates the preferred form of "no-op" (an instruction that does nothing). This is shown as an example with the *OR Immediate* instruction.

Extended mnemonics are provided that use the *OR* and *NOR* instructions to copy the contents of one register to another, with and without complementing. These are shown as examples with the two instructions.

See Appendix C, "Assembler Extended Mnemonics" on page 133 for additional extended mnemonics.

AND Immediate D-form

andi. RA,RS,UI

[Power mnemonic: andil.]

28	RS	RA	UI
0	6	11	16 31

RA ← (RS) & ($^{48}0$ || UI)

The contents of register RS is ANDed with $^{48}0$ || UI and the result is placed into register RA.

Special Registers Altered:
CR0

AND Immediate Shifted D-form

andis. RA,RS,UI

[Power mnemonic: andiu.]

29	RS	RA	UI
0	6	11	16 31

RA ← (RS) & ($^{32}0$ || UI || $^{16}0$)

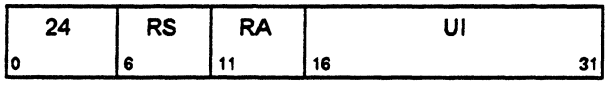
The contents of register RS is ANDed with $^{32}0$ || UI || $^{16}0$ and the result is placed into register RA.

Special Registers Altered:
CR0

OR Immediate D-form

ori RA,RS,UI

[Power mnemonic: oril]



RA ← (RS) | ($^{48}0$ || UI)

The contents of register RS is ORed with $^{48}0$ || UI and the result is placed into register RA.

The preferred "no-op" (an instruction that does nothing) is:

```
ori  0,0,0
```

Special Registers Altered:
None

Extended Mnemonics:

Example of extended mnemonics for *OR Immediate*:

Extended:	*Equivalent to:*
nop	ori 0,0,0

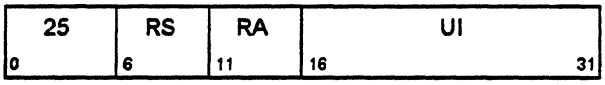
Engineering Note

It is desirable for implementations to make the preferred form of "no-op" execute quickly, since this form should be used by compilers.

OR Immediate Shifted D-form

oris RA,RS,UI

[Power mnemonic: oriu]



RA ← (RS) | ($^{32}0$ || UI || $^{16}0$)

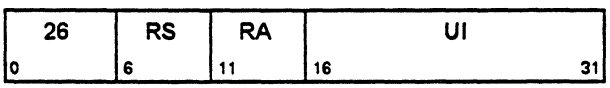
The contents of register RS is ORed with $^{32}0$ || UI || $^{16}0$ and the result is placed into register RA.

Special Registers Altered:
None

XOR Immediate D-form

xori RA,RS,UI

[Power mnemonic: xoril]



RA ← (RS) ⊕ ($^{48}0$ || UI)

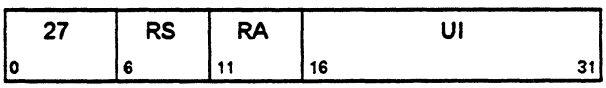
The contents of register RS is XORed with $^{48}0$ || UI and the result is placed into register RA.

Special Registers Altered:
None

XOR Immediate Shifted D-form

xoris RA,RS,UI

[Power mnemonic: xoriu]



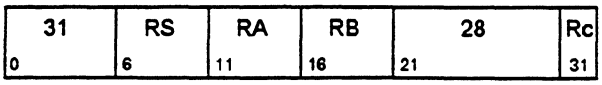
RA ← (RS) ⊕ ($^{32}0$ || UI || $^{16}0$)

The contents of register RS is XORed with $^{32}0$ || UI || $^{16}0$ and the result is placed into register RA.

Special Registers Altered:
None

AND X-form

and	RA,RS,RB	(Rc=0)
and.	RA,RS,RB	(Rc=1)



RA ← (RS) & (RB)

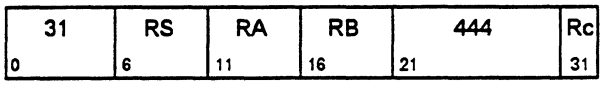
The contents of register RS is ANDed with the contents of register RB and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

OR X-form

or	RA,RS,RB	(Rc=0)
or.	RA,RS,RB	(Rc=1)



RA ← (RS) | (RB)

The contents of register RS is ORed with the contents of register RB and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

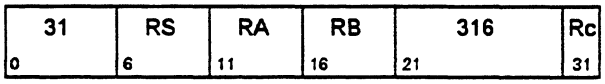
Extended Mnemonics:

Example of extended mnemonics for *OR*:

Extended:		*Equivalent to:*	
mr	Rx,Ry	or	Rx,Ry,Ry

XOR X-form

xor	RA,RS,RB	(Rc=0)
xor.	RA,RS,RB	(Rc=1)



RA ← (RS) ⊕ (RB)

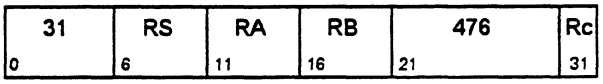
The contents of register RS is XORed with the contents of register RB and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

NAND X-form

nand	RA,RS,RB	(Rc=0)
nand.	RA,RS,RB	(Rc=1)



RA ← ¬((RS) & (RB))

The contents of register RS is ANDed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:

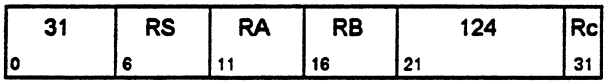
CR0 (if Rc=1)

Programming Note

nand or ***nor*** with RA=RB can be used to obtain the one's complement.

NOR X-form

nor RA,RS,RB (Rc=0)
nor. RA,RS,RB (Rc=1)



RA ← ¬((RS) | (RB))

The contents of register RS is ORed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

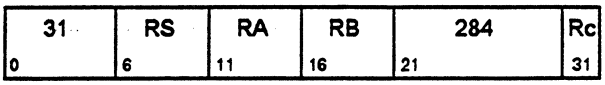
Extended Mnemonics:

Example of extended mnemonics for *NOR*:

Extended:		*Equivalent to:*	
not	Rx,Ry	nor	Rx,Ry,Ry

Equivalent X-form

eqv RA,RS,RB (Rc=0)
eqv. RA,RS,RB (Rc=1)



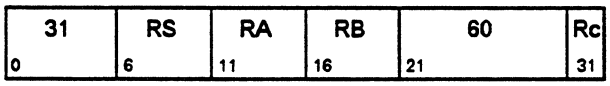
RA ← (RS) ≡ (RB)

The contents of register RS is XORed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

AND with Complement X-form

andc RA,RS,RB (Rc=0)
andc. RA,RS,RB (Rc=1)



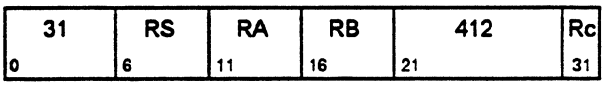
RA ← (RS) & ¬(RB)

The contents of register RS is ANDed with the complement of the contents of register RB and the result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

OR with Complement X-form

orc RA,RS,RB (Rc=0)
orc. RA,RS,RB (Rc=1)



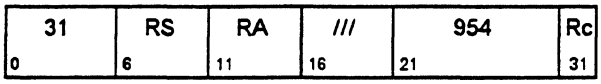
RA ← (RS) | ¬(RB)

The contents of register RS is ORed with the complement of the contents of register RB and the result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

Extend Sign Byte X-form

extsb RA,RS (Rc=0)
extsb. RA,RS (Rc=1)



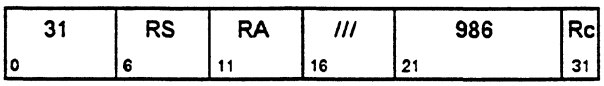
$s \leftarrow (RS)_{56}$
$RA_{56:63} \leftarrow (RS)_{56:63}$
$RA_{0:55} \leftarrow {}^{56}s$

$(RS)_{56:63}$ are placed into $RA_{56:63}$. Bit 56 of register RS is placed into $RA_{0:55}$.

Special Registers Altered:
CR0 (if Rc=1)

Extend Sign Word X-form

extsw RA,RS (Rc=0)
extsw. RA,RS (Rc=1)



$s \leftarrow (RS)_{32}$
$RA_{32:63} \leftarrow (RS)_{32:63}$
$RA_{0:31} \leftarrow {}^{32}s$

$(RS)_{32:63}$ are placed into $RA_{32:63}$. Bit 32 of register RS is placed into $RA_{0:31}$.

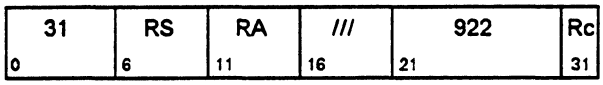
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:
CR0 (if Rc=1)

Extend Sign Halfword X-form

extsh RA,RS (Rc=0)
extsh. RA,RS (Rc=1)

[Power mnemonics: exts, exts.]



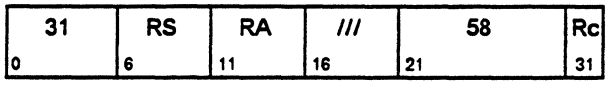
$s \leftarrow (RS)_{48}$
$RA_{48:63} \leftarrow (RS)_{48:63}$
$RA_{0:47} \leftarrow {}^{48}s$

$(RS)_{48:63}$ are placed into $RA_{48:63}$. Bit 48 of register RS is placed into $RA_{0:47}$.

Special Registers Altered:
CR0 (if Rc=1)

Count Leading Zeros Doubleword X-form

cntlzd RA,RS (Rc=0)
cntlzd. RA,RS (Rc=1)




```
n ← 0
do while n < 64
  if (RS)n = 1 then leave
  n ← n + 1
RA ← n
```

A count of the number of consecutive zero bits starting at bit 0 of register RS is placed into RA. This number ranges from 0 to 64, inclusive.

If Rc=1, CR Field 0 is set to reflect the result.

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.

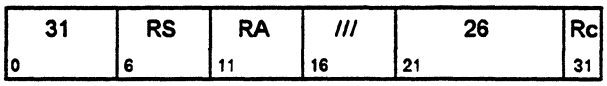
Special Registers Altered:

CR0 (if Rc=1)

Count Leading Zeros Word X-form

cntlzw RA,RS (Rc=0)
cntlzw. RA,RS (Rc=1)

[Power mnemonics: cntlz, cntlz.]




```
n ← 32
do while n < 64
  if (RS)n = 1 then leave
  n ← n + 1
RA ← n - 32
```

A count of the number of consecutive zero bits starting at bit 32 of of register RS is placed into RA. This number ranges from 0 to 32, inclusive.

If Rc=1, CR Field 0 is set to reflect the result.

Special Registers Altered:

CR0 (if Rc=1)

Programming Note

For both *Count Leading Zeros* instructions, if Rc=1 then LT is set to zero in CR Field 0.

3.3.13 Fixed-Point Rotate and Shift Instructions

The Fixed-Point Processor performs rotation operations on data from a GPR and returns the result, or a portion of the result, to a GPR.

The rotation operations rotate a 64-bit quantity left by a specified number of bit positions. Bits that exit from position 0 enter at position 63.

Two types of rotation operation are supported.

For the first type, denoted $rotate_{64}$ or $ROTL_{64}$, the value rotated is the given 64-bit value. The $rotate_{64}$ operation is used to rotate a given 64-bit quantity.

For the second type, denoted $rotate_{32}$ or $ROTL_{32}$, the value rotated consists of two copies of bits 32:63 of the given 64-bit value, one copy in bits 0:31 and the other in bits 32:63. The $rotate_{32}$ operation is used to rotate a given 32-bit quantity.

The *Rotate* and *Shift* instructions employ a mask generator. The mask is 64 bits long, and consists of 1-bits from a start bit, *mstart*, through and including a stop bit, *mstop*, and 0-bits elsewhere. The values of *mstart* and *mstop* range from zero to 63. If *mstart* > *mstop*, the 1-bits wrap around from position 63 to position 0. Thus the mask is formed as follows:

```
if mstart ≤ mstop then
   mask[mstart:mstop] = ones
   mask[all other bits] = zeros
else
   mask[mstart:63] = ones
   mask[0:mstop] = ones
   mask[all other bits] = zeros
```

There is no way to specify an all-zero mask.

For instructions that use the $rotate_{32}$ operation, the mask start and stop positions are always in the low-order 32 bits of the register.

The use of the mask is described in following sections.

If Rc=1, the *Rotate* and *Shift* instructions set CR Field 0 according to the contents of register RA at the completion of the instruction. *Rotate* and *Shift* instructions do not change the OV and SO bits. *Rotate* and *Shift* instructions, except algebraic right shifts, do not change the CA bit.

Extended mnemonics for rotates and shifts

The *Rotate and Shift* instructions, while powerful, can be complicated to code (they have up to five operands). A set of extended mnemonics is provided that allow simpler coding of often-used functions such as clearing the leftmost or rightmost bits of a register, left justifying or right justifying an arbitrary field, and simple rotates and shifts. Some of these are shown as examples with the *Rotate* instructions. See Appendix C, "Assembler Extended Mnemonics" on page 133 for additional extended mnemonics.

3.3.13.1 Fixed-Point Rotate Instructions

These instructions rotate the contents of a register. The result of the rotation is

- Inserted into the target register under control of a mask (if a mask bit is 1 the associated bit of the rotated data is placed into the target register, and if the mask bit is 0 the associated bit in the target register remains unchanged); or
- ANDed with a mask before being placed into the target register.

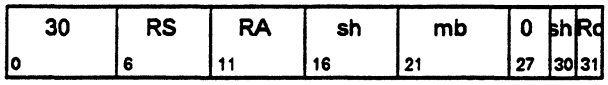
The *Rotate Left* instructions allow right-rotation of the contents of a register to be performed (in concept) by a left-rotation of 64—N, where N is the number of bits by which to rotate right. They allow right-rotation of the contents of the low-order 32 bits of a register to be performed (in concept) by a left-rotation of 32—N, where N is the number of bits by which to rotate right.

Architecture Note

For MD-form and MDS-form instructions, the MB and ME fields are used in permuted rather than sequential order because this is easier for the processor. Permuting the MB field permits the processor to obtain the low-order five bits of the MB value from the same place for all instructions having an MB field (M-form and MD-form instructions). Permuting the ME field permits the processor to treat bits 21:26 of all MD-form instructions uniformly.

Rotate Left Doubleword Immediate then Clear Left MD-form

rldicl RA,RS,SH,MB (Rc=0)
rldicl. RA,RS,SH,MB (Rc=1)




```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, 63)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left SH bits. 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 register 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.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Rotate Left Doubleword Immediate then Clear Left*:

Extended:		*Equivalent to:*	
extrdi	Rx,Ry,n,b	rldicl	Rx,Ry,b+n,64−n
srdi	Rx,Ry,n	rldicl	Rx,Ry,64−n,n
clrldi	Rx,Ry,n	rldicl	Rx,Ry,0,n

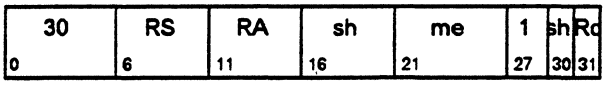
Programming Note

rldicl can be used to extract an *n*-bit field, that starts at bit position *b* in register RS, right-justified into register RA (clearing the remaining 64−*n* bits of RA), by setting SH=*b*+*n* and MB=64−*n*. It can be used to rotate the contents of a register left (right) by *n* bits, by setting SH=*n* (64−*n*) and MB=0. It can be used to shift the contents of a register right by *n* bits, by setting SH=64−*n* and MB=*n*. It can be used to clear the high-order *n* bits of a register, by setting SH=0 and MB=*n*.

Extended mnemonics are provided for all of these uses: see Appendix C, "Assembler Extended Mnemonics" on page 133.

Rotate Left Doubleword Immediate then Clear Right MD-form

rldicr RA,RS,SH,ME (Rc=0)
rldicr. RA,RS,SH,ME (Rc=1)




```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
e ← me5 || me0:4
m ← MASK(0, e)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left SH bits. 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 register 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.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Rotate Left Doubleword Immediate then Clear Right*:

Extended:		*Equivalent to:*	
extldi	Rx,Ry,n,b	rldicr	Rx,Ry,b,n−1
sldi	Rx,Ry,n	rldicr	Rx,Ry,n,63−n
clrrdi	Rx,Ry,n	rldicr	Rx,Ry,0,63−n

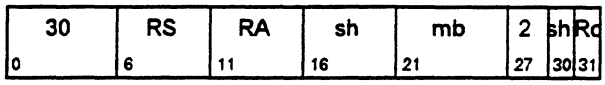
Programming Note

rldicr can be used to extract an *n*-bit field, that starts at bit position *b* in register RS, left-justified into register RA (clearing the remaining 64−*n* bits of RA), by setting SH=*b* and ME=*n*−1. It can be used to rotate the contents of a register left (right) by *n* bits, by setting SH=*n* (64−*n*) and ME=63. It can be used to shift the contents of a register left by *n* bits, by setting SH=*n* and ME=63−*n*. It can be used to clear the low-order *n* bits of a register, by setting SH=0 and ME=63−*n*.

Extended mnemonics are provided for all of these uses (some devolve to ***rldicl***): see Appendix C, "Assembler Extended Mnemonics" on page 133.

Rotate Left Doubleword Immediate then Clear MD-form

rldic RA,RS,SH,MB (Rc=0)
rldic. RA,RS,SH,MB (Rc=1)




```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, ¬n)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left SH bits. 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 register 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.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Doubleword Immediate then Clear*:

Extended:		*Equivalent to:*	
clrlsldi	Rx,Ry,b,n	rldic	Rx,Ry,n,b—n

Programming Note

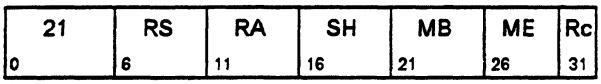
rldic can be used to clear the high-order *b* bits of the contents of a register and then shift the result left by *n* bits by setting SH=*n* and MB=*b—n*. It can be used to clear the high-order *n* bits of a register, by setting SH=0 and MB=*n*.

Extended mnemonics are provided for both of these uses (the second devolves to ***rldicl***): see Appendix C, "Assembler Extended Mnemonics" on page 133.

Rotate Left Word Immediate then AND with Mask M-form

rlwinm RA,RS,SH,MB,ME (Rc=0)
rlwinm. RA,RS,SH,MB,ME (Rc=1)

[Power mnemonics: rlinm, rlinm.]




```
n ← SH
r ← ROTL32((RS)32:63, n)
m ← MASK(MB+32, ME+32)
RA ← r & m
```

The contents of register RS are rotated$_{32}$ left SH bits. A mask is generated having 1-bits from bit MB through bit ME and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Rotate Left Word Immediate then AND with Mask*:

Extended:		*Equivalent to:*
extlwi	Rx,Ry,n,b	rlwinm Rx,Ry,b,0,n—1
srwi	Rx,Ry,n	rlwinm Rx,Ry,32—n,n,31
clrrwi	Rx,Ry,n	rlwinm Rx,Ry,0,0,31—n

Programming Note

Let RSL represent the low-order 32 bits of register RS, with the bits numbered from 0 through 31.

rlwinm can be used to extract an *n*-bit field, that starts at bit position *b* in RSL, right-justified into the low-order 32 bits of register RA (clearing the remaining 32−*n* bits of the low-order 32 bits of RA), by setting SH=*b*+*n*, MB=32−*n*, and ME=31. It can be used to extract an *n*-bit field, that starts at bit position *b* in RSL, left-justified into the low-order 32 bits of register RA (clearing the remaining 32−*n* bits of the low-order 32 bits of RA), by setting SH=*b*, MB = 0, and ME=*n*−1. It can be used to rotate the contents of the low-order 32 bits of a register left (right) by *n* bits, by setting SH=*n* (32−*n*), MB=0, and ME=31. It can be used to shift the contents of the low-order 32 bits of a register right by *n* bits, by setting SH=32−*n*, MB=*n*, and ME=31. It can be used to clear the high-order *b* bits of the low-order 32 bits of the contents of a register and then shift the result left by *n* bits by setting SH=*n*, MB=*b*−*n* and ME=31−*n*. It can be used to clear the low-order *n* bits of the low-order 32 bits of a register, by setting SH=0, MB=0, and ME=31−*n*.

For all the uses given above, the high-order 32 bits of register RA are cleared.

Extended mnemonics are provided for all of these uses: see Appendix C, "Assembler Extended Mnemonics" on page 133.

Rotate Left Doubleword then Clear Left MDS-form

rldcl RA,RS,RB,MB (Rc=0)
rldcl. RA,RS,RB,MB (Rc=1)

30	RS	RA	RB	mb	8	Rc
0	6	11	16	21	27	31

```
n ← (RB)58:63
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, 63)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left the number of bits specified by (RB)$_{58:63}$. 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 register 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.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Doubleword then Clear Left*:

Extended:		*Equivalent to:*	
rotld	Rx,Ry,Rz	rldcl	Rx,Ry,Rz,0

Programming Note

rldcl can be used to extract an *n*-bit field, that starts at variable bit position *b* in register RS, right-justified into register RA (clearing the remaining 64−*n* bits of RA), by setting RB$_{58:63}$=*b*+*n* and MB=64−*n*. It can be used to rotate the contents of a register left (right) by variable *n* bits by setting RB$_{58:63}$=*n* (64−*n*) and MB=0.

Extended mnemonics are provided for some of these uses: see Appendix C, "Assembler Extended Mnemonics" on page 133.

Rotate Left Doubleword then Clear Right MDS-form

rldcr RA,RS,RB,ME (Rc=0)
rldcr. RA,RS,RB,ME (Rc=1)

30	RS	RA	RB	me	9	Rc
0	6	11	16	21	27	31

n ← $(RB)_{58:63}$
r ← $ROTL_{64}((RS), n)$
e ← $me_5 \| me_{0:4}$
m ← MASK(0, e)
RA ← r & m

The contents of register RS are $rotated_{64}$ left the number of bits specified by $(RB)_{58:63}$. 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 register 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.

Special Registers Altered:
CR0 (if Rc=1)

Programming Note

rldcr can be used to extract an *n*-bit field, that starts at variable bit position *b* in register RS, left-justified into register RA (clearing the remaining 64−*n* bits of RA), by setting $RB_{58:63}=b$ and ME=*n*−1. It can be used to rotate the contents of a register left (right) by variable *n* bits by setting $RB_{58:63}=n$ (64−*n*) and ME=63.

Extended mnemonics are provided for some of these uses (some devolve to ***rldcl***) see Appendix C, "Assembler Extended Mnemonics" on page 133.

Rotate Left Word then AND with Mask M-form

rlwnm RA,RS,RB,MB,ME (Rc=0)
rlwnm. RA,RS,RB,MB,ME (Rc=1)

[Power mnemonics: rlnm, rlnm.]

23	RS	RA	RB	MB	ME	Rc
0	6	11	16	21	26	31

n ← $(RB)_{59:63}$
r ← $ROTL_{32}((RS)_{32:63}, n)$
m ← MASK(MB+32, ME+32)
RA ← r & m

The contents of register RS are $rotated_{32}$ left the number of bits specified by $(RB)_{59:63}$. A mask is generated having 1-bits from bit MB through bit ME and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Word then AND with Mask*:

Extended:	*Equivalent to:*
rotlw Rx,Ry,Rz	rlwnm Rx,Ry,Rz,0,31

Programming Note

Let RSL represent the low-order 32 bits of register RS, with the bits numbered from 0 through 31.

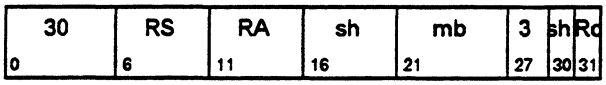
rlwnm can be used to extract an *n*-bit field, that starts at variable bit position *b* in RSL, right-justified into the low-order 32 bits of register RA (clearing the remaining 32−*n* bits of the low-order 32 bits of RA), by setting $RB_{59:63}=b+n$, MB=32−*n*, and ME=31. It can be used to extract an *n*-bit field, that starts at variable bit position *b* in RSL, left-justified into the low-order 32 bits of register RA (clearing the remaining 32−*n* bits of the low-order 32 bits of RA), by setting $RB_{59:63}=b$, MB = 0, and ME=*n*−1. It can be used to rotate the contents of the low-order 32 bits of a register left (right) by variable *n* bits, by setting $RB_{59:63}=n$ (32−*n*), MB=0, and ME=31.

For all the uses given above, the high-order 32 bits of register RA are cleared.

Extended mnemonics are provided for some of these uses: see Appendix C, "Assembler Extended Mnemonics" on page 133.

Rotate Left Doubleword Immediate then Mask Insert MD-form

rldimi RA,RS,SH,MB (Rc=0)
rldimi. RA,RS,SH,MB (Rc=1)




```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, ¬n)
RA ← r&m | (RA)&¬m
```

The contents of register RS are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit MB through bit 63−SH, and 0-bits elsewhere. The rotated data is inserted into register RA under control of the generated mask.

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.

Special Registers Altered:
CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Doubleword Immediate then Mask Insert*:

Extended:	*Equivalent to:*
insrdi Rx,Ry,n,b	rldimi Rx,Ry,64−(b+n),b

Programming Note

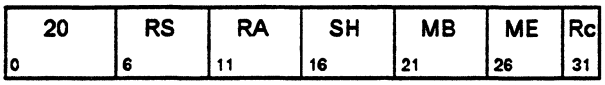
rldimi can be used to insert an *n*-bit field, that is right-justified in register RS, into register RA starting at bit position *b*, by setting SH=64−(*b*+*n*) and MB=*b*.

An extended mnemonic is provided for this use: see Appendix C, "Assembler Extended Mnemonics" on page 133.

Rotate Left Word Immediate then Mask Insert M-form

rlwimi RA,RS,SH,MB,ME (Rc=0)
rlwimi. RA,RS,SH,MB,ME (Rc=1)

[Power mnemonics: rlimi, rlimi.]




```
n ← SH
r ← ROTL32((RS)32:63, n)
m ← MASK(MB+32, ME+32)
RA ← r&m | (RA)&¬m
```

The contents of register RS are rotated$_{32}$ left SH bits. A mask is generated having 1-bits from bit MB through bit ME and 0-bits elsewhere. The rotated data is inserted into register RA under control of the generated mask.

Special Registers Altered:
CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Word Immediate then Mask Insert*:

Extended:	*Equivalent to:*
inslwi Rx,Ry,n,b	rlwimi Rx,Ry,32−b,b,b+n−1

Programming Note

Let RAL represent the low-order 32 bits of register RA, with the bits numbered from 0 through 31.

rlwimi can be used to insert an *n*-bit field, that is left-justified in the low-order 32 bits of register RS, into RAL starting at bit position *b*, by setting SH=32−*b*, MB=*b*, and ME=(*b*+*n*)−1. It can be used to insert an *n*-bit field, that is right-justified in the low-order 32 bits of register RS, into RAL starting at bit position *b*, by setting SH=32−(*b*+*n*), MB=*b*, and ME=(*b*+*n*)−1.

Extended mnemonics are provided for both of these uses: see Appendix C, "Assembler Extended Mnemonics" on page 133.

3.3.13.2 Fixed-Point Shift Instructions

The instructions in this section perform left and right shifts.

Extended mnemonics for shifts

Immediate-form logical (unsigned) shift operations are obtained by specifying appropriate masks and shift values for certain *Rotate* instructions. A set of extended mnemonics is provided to make coding of such shifts simpler and easier to understand. and simple rotates and shifts. Some of these are shown as examples with the *Rotate* instructions. See Appendix C, "Assembler Extended Mnemonics" on page 133 for additional extended mnemonics.

Programming Note

Any *Shift Right Algebraic* instruction, followed by ***addze***, can be used to divide quickly by 2^N. The setting of the CA bit by the *Shift Right Algebraic* instructions is independent of mode.

Programming Note

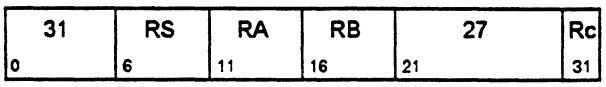
Multiple-precision shifts can be programmed as shown in Appendix E.2, "Multiple-Precision Shifts" on page 156.

Engineering Note

The instructions intended for use with 32-bit data are shown as doing a rotate_{32} operation. This is strictly necessary only for setting the CA bit for ***srawi*** and ***sraw***. ***slw*** and ***srw*** could do a rotate_{64} operation if that is easier.

Shift Left Doubleword X-form

sld RA,RS,RB (Rc=0)
sld. RA,RS,RB (Rc=1)




```
n ← (RB)58:63
r ← ROTL64((RS), n)
if (RB)57 = 0 then
     m ← MASK(0, 63-n)
else m ← 640
RA ← r & m
```

The contents of register RS are shifted left the number of bits specified by $(RB)_{57:63}$. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The result is placed into register RA. Shift amounts from 64 to 127 give a zero result.

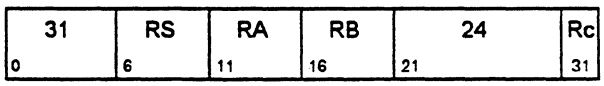
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:
CR0 (if Rc=1)

Shift Left Word X-form

slw RA,RS,RB (Rc=0)
slw. RA,RS,RB (Rc=1)

[Power mnemonics: sl, sl.]




```
n ← (RB)59:63
r ← ROTL32((RS)32:63, n)
if (RB)58 = 0 then
     m ← MASK(32, 63-n)
else m ← 640
RA ← r & m
```

The contents of the low-order 32 bits of register RS are shifted left the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into $RA_{32:63}$. $RA_{0:31}$ are set to zero. Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:
CR0 (if Rc=1)

Shift Right Doubleword X-form

srd RA,RS,RB (Rc=0)
srd. RA,RS,RB (Rc=1)

31	RS	RA	RB	539	Rc
0	6	11	16	21	31

```
n ← (RB)58:63
r ← ROTL64((RS), 64-n)
if (RB)57 = 0 then
     m ← MASK(n, 63)
else m ← 64 0
RA ← r & m
```

The contents of register RS are shifted right the number of bits specified by $(RB)_{57:63}$. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The result is placed into register RA. Shift amounts from 64 to 127 give a zero result.

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.

Special Registers Altered:
CR0 (if Rc=1)

Shift Right Word X-form

srw RA,RS,RB (Rc=0)
srw. RA,RS,RB (Rc=1)

[Power mnemonics: sr, sr.]

31	RS	RA	RB	536	Rc
0	6	11	16	21	31

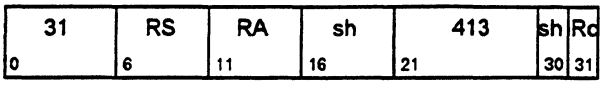
```
n ← (RB)59:63
r ← ROTL32((RS)32:63, 64-n)
if (RB)58 = 0 then
     m ← MASK(n+32, 63)
else m ← 64 0
RA ← r & m
```

The contents of the low-order 32 bits of register RS are shifted right the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into $RA_{32:63}$. $RA_{0:31}$ are set to zero. Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:
CR0 (if Rc=1)

Shift Right Algebraic Doubleword Immediate XS-form

sradi RA,RS,SH (Rc=0)
sradi. RA,RS,SH (Rc=1)



$n \leftarrow sh_5 \,||\, sh_{0:4}$
$r \leftarrow ROTL_{64}((RS), 64-n)$
$m \leftarrow MASK(n, 63)$
$s \leftarrow (RS)_0$
$RA \leftarrow r\&m \mid (^{64}s)\&\neg m$
$CA \leftarrow s \;\&\; ((r\&\neg m)\neq 0)$

The contents of register RS are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 0 of RS is replicated to fill the vacated positions on the left. The result is placed into register RA. CA is set to 1 if (RS) is negative and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to be set equal to (RS), and CA to be set to 0.

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.

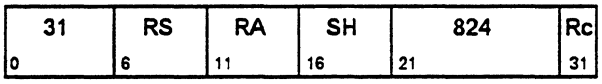
Special Registers Altered:

CA
CR0 (if Rc=1)

Shift Right Algebraic Word Immediate X-form

srawi RA,RS,SH (Rc=0)
srawi. RA,RS,SH (Rc=1)

[Power mnemonics: srai, srai.]



$n \leftarrow SH$
$r \leftarrow ROTL_{32}((RS)_{32:63}, 64-n)$
$m \leftarrow MASK(n+32, 63)$
$s \leftarrow (RS)_{32}$
$RA \leftarrow r\&m \mid (^{64}s)\&\neg m$
$CA \leftarrow s \;\&\; ((r\&\neg m)_{32:63}\neq 0)$

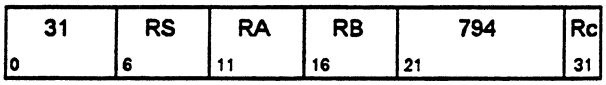
The contents of the low-order 32 bits of register RS are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 32 of RS is replicated to fill the vacated positions on the left. The 32-bit result is placed into $RA_{32:63}$. Bit 32 of RS is replicated to fill $RA_{0:31}$. CA is set to 1 if the low-order 32 bits of (RS) contain a negative number and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to receive $EXTS((RS)_{32:63})$, and CA to be set to 0.

Special Registers Altered:

CA
CR0 (if Rc=1)

Shift Right Algebraic Doubleword X-form

srad RA,RS,RB (Rc=0)
srad. RA,RS,RB (Rc=1)




```
n ← (RB)58:63
r ← ROTL64((RS), 64-n)
if (RB)57 = 0 then
     m ← MASK(n, 63)
else m ← 64 0
s ← (RS)0
RA ← r&m | (64s)&¬m
CA ← s & ((r&¬m)≠0)
```

The contents of register RS are shifted right the number of bits specified by $(RB)_{57:63}$. Bits shifted out of position 63 are lost. Bit 0 of RS is replicated to fill the vacated positions on the left. The result is placed into register RA. CA is set to 1 if (RS) is negative and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to be set equal to (RS), and CA to be set to 0. Shift amounts from 64 to 127 give a result of 64 sign bits in RA, and cause CA to receive the sign bit of (RS).

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.

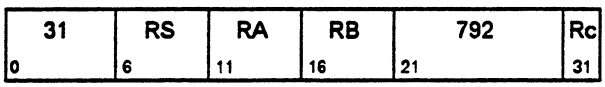
Special Registers Altered:

CA
CR0 (if Rc=1)

Shift Right Algebraic Word X-form

sraw RA,RS,RB (Rc=0)
sraw. RA,RS,RB (Rc=1)

[Power mnemonics: sra, sra.]




```
n ← (RB)59:63
r ← ROTL32((RS)32:63, 64-n)
if (RB)58 = 0 then
     m ← MASK(n+32, 63)
else m ← 64 0
s ← (RS)32
RA ← r&m | (64s)&¬m
CA ← s & ((r&¬m)32:63≠0)
```

The contents of the low-order 32 bits of register RS are shifted right the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 63 are lost. Bit 32 of RS is replicated to fill the vacated positions on the left. The 32-bit result is placed into $RA_{32:63}$. Bit 32 of RS is replicated to fill $RA_{0:31}$. CA is set to 1 if the low-order 32 bits of (RS) contain a negative number and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to receive $EXTS((RS)_{32:63})$, and CA to be set to 0. Shift amounts from 32 to 63 give a result of 64 sign bits, and cause CA to receive the sign bit of $(RS)_{32:63}$.

Special Registers Altered:

CA
CR0 (if Rc=1)

3.3.14 Move To/From System Register Instructions

Extended mnemonics

A set of extended mnemonics is provided for the ***mtspr*** and ***mfspr*** instructions so that they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the two instructions. See Appendix C, "Assembler Extended Mnemonics" on page 133 for additional extended mnemonics.

Move To Special Purpose Register XFX-form

mtspr SPR,RS

31	RS	spr	467	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
if length(SPREG(n)) = 64 then
  SPREG(n) ← (RS)
else
  SPREG(n) ← (RS)32:63(0:31)
```

The SPR field denotes a Special Purpose Register, encoded as shown in the table below. The contents of register RS are placed into the designated Special Purpose Register. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

decimal	SPR* $spr_{5:9}$ $spr_{0:4}$	Register name
1	00000 00001	XER
8	00000 01000	LR
9	00000 01001	CTR

* Note that the order of the two 5-bit halves of the SPR number is reversed.

Additional values of the SPR field are defined in Book III, *PowerPC Operating Environment Architecture*, and others may be defined in Book IV, *PowerPC Implementation Features* for the implementation. If the SPR field contains any value other than one of these implementation-specific values or one of the values shown above or in Book III, the instruction form is invalid. For an invalid instruction form in which $spr_0 = 1$, the system privileged instruction error handler may be invoked instead of the system illegal instruction error handler.

Special Registers Altered:

See above

Extended Mnemonics:

Examples of extended mnemonics for *Move To Special Purpose Register*:

Extended:	*Equivalent to:*
mtxer Rx	mtspr 1,Rx
mtlr Rx	mtspr 8,Rx
mtctr Rx	mtspr 9,Rx

Compiler and Assembler Note

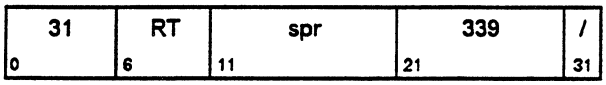
For the ***mtspr*** and ***mfspr*** instructions, the SPR number coded in assembler language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16:20 of the instruction and the low-order 5 bits in bits 11:15. This maintains compatibility with Power SPR encodings, in which these two instructions had only a 5-bit SPR field occupying bits 11:15.

Compatibility Note

For a discussion of Power compatibility with respect to SPR numbers not shown in the instruction descriptions for ***mtspr*** and ***mfspr***, please refer to Appendix G, "Incompatibilities with the Power Architecture" on page 165. For compatibility with future versions of this architecture, only SPR numbers discussed in these instruction descriptions should be used.

Move From Special Purpose Register
XFX-form

mfspr RT,SPR




```
n ← spr5:9 || spr0:4
if length(SPREG(n)) = 64 then
  RT ← SPREG(n)
else
  RT ← 32θ || SPREG(n)
```

The SPR field denotes a Special Purpose Register, encoded as shown in the table below. The contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

	SPR*	Register
decimal	$spr_{5:9}$ $spr_{0:4}$	name
1	00000 00001	XER
8	00000 01000	LR
9	00000 01001	CTR

* Note that the order of the two 5-bit halves of the SPR number is reversed.

Additional values of the SPR field are defined in Book III, *PowerPC Operating Environment Architecture*, and others may be defined in Book IV, *PowerPC Implementation Features* for the implementation. If the SPR field contains any value other than one of these implementation-specific values or one of the values shown above or in Book III, the instruction form is invalid. For an invalid instruction form in which $spr_0 = 1$, the system privileged instruction error handler may be invoked instead of the system illegal instruction error handler.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Move From Special Purpose Register*:

Extended:	*Equivalent to:*
mfxer Rx	mfspr Rx,1
mflr Rx	mfspr Rx,8
mfctr Rx	mfspr Rx,9

Compiler/Assembler/Compatibility Notes

See the Notes that appear with ***mtspr***.

Move To Condition Register Fields XFX-form

mtcrf FXM,RS

31	RS	/	FXM	/	144	/
0	6	11	12	20	21	31

$mask \leftarrow {}^{4}(FXM_0) \parallel {}^{4}(FXM_1) \parallel \ldots {}^{4}(FXM_7)$
$CR \leftarrow ((RS)_{32:63} \;\&\; mask) \mid (CR \;\&\; \neg mask)$

The contents of bits 32:63 of register RS are placed into the Condition Register under control of the field mask specified by FXM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0-7. If FXM(i) = 1 then CR field i (CR bits 4xi through 4xi+3) is set to the contents of the corresponding field of the low-order 32 bits of RS.

Special Registers Altered:
CR fields selected by mask

Programming Note

Updating a proper subset of the eight fields of the Condition Register may have substantially poorer performance on some implementations than updating all of the fields.

Move to Condition Register from XER X-form

mcrxr BF

31	BF	//	///	///	512	/
0	6	9	11	16	21	31

$CR_{4\times BF:4\times BF+3} \leftarrow XER_{0:3}$
$XER_{0:3} \leftarrow 0b0000$

The contents of $XER_{0:3}$ are copied into the Condition Register field designated by BF. $XER_{0:3}$ is set to zero.

Special Registers Altered:
CR $XER_{0:3}$

Move From Condition Register X-form

mfcr RT

31	RT	///	///	19	/
0	6	11	16	21	31

$RT \leftarrow {}^{32}0 \parallel CR$

The contents of the Condition Register are placed into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:
None

Chapter 4. Floating-Point Processor

4.1 Floating-Point Processor Overview

The Floating-Point Processor provides high performance execution of floating-point operations. Instructions are provided to perform arithmetic, conversion, comparison, and other operations in floating-point registers, and to move floating-point data between storage and these registers. Instructions in the first group are called "arithmetic instructions," and instructions in the second group are called "storage access instructions." Instructions are also provided that manipulate the Floating-Point Status and Control Register.

This architecture provides for the processor to implement a floating-point system as defined in ANSI/IEEE Standard 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic" (hereafter referred to as "the IEEE standard"), but has a dependency on supporting software to be in "conformance" with that standard. All floating-point operations conform to that standard, except if software sets the Floating-Point Non-IEEE Mode (NI) bit in the Floating-Point Status and Control Register to 1 (see page 86), in which case floating-point operations do not necessarily conform to that standard.

A floating-point number consists of a signed exponent and a signed significand. The quantity expressed by

this number is the product of the significand and the number $2^{exponent}$. Encodings are provided in the data format to represent finite numeric values, ±Infinity, and values which 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 such things as uninitialized variables and can be produced by certain invalid operations.

There is one class of exceptional events which occur during instruction execution which are unique to the Floating-Point Processor:

- Floating-Point Exception

Floating-point exceptions are signalled with bits set in the Floating-Point Status and Control Register (FPSCR). They can cause the system floating-point enabled exception error handler to be invoked, precisely or imprecisely, if the proper control bits are set.

Floating-Point Exceptions

The following floating-point exceptions are detected by the processor:

- Invalid Operation Exception (VX)
  - SNaN (VXSNAN)
  - Infinity−Infinity (VXISI)
  - Infinity÷Infinity (VXIDI)
  - Zero÷Zero (VXZDZ)
  - Infinity×Zero (VXIMZ)
  - Invalid Compare (VXVC)
  - Software Request (VXSOFT)
  - Invalid Square Root (VXSQRT)
  - Invalid Integer Convert (VXCVI)
- Zero Divide Exception (ZX)
- Overflow Exception (OX)
- Underflow Exception (UX)
- Inexact Exception (XX)

Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. See Section 4.2.2, "Floating-Point Status and Control Register" on page 85, for a description of these exception and enable bits, and Section 4.4, "Floating-Point Exceptions" on page 91, for a detailed discussion of floating-point exceptions, including the effects of the enable bits.

4.2 Floating-Point Processor Registers

4.2.1 Floating-Point Registers

Implementations of this architecture provide 32 floating-point registers (FPR). The floating-point instruction formats provide a 5-bit field for specifying the FPRs to be used in the execution of the instruction. The FPRs are numbered 0-31.

Each FPR contains 64 bits which support the floating-point double format. Every instruction that interprets the contents of an FPR as a floating-point value uses the floating-point double format for this interpretation.

Every floating-point arithmetic instruction operates on data located in FPRs and, with the exception of the *Compare* instructions, places the result value into an FPR. Status information is placed into the Floating-Point Status and Control Register and in some cases into the Condition Register.

Load and store double instructions are provided that transfer 64 bits of data between storage and the FPRs in the Floating-Point Processor with no conversion. Load single instructions are provided to transfer and convert floating-point values in floating-point single format from storage to the same value in floating-point double format in the FPRs. Store single instructions are provided to transfer and convert floating-point values in floating-point double format from the FPRs to the same value in floating-point single format in storage.

Single- and double-precision arithmetic instructions accept values from the FPRs in double format. For single-precision arithmetic instructions, all input values must be representable in single format: if they are not, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the Condition Register (if Rc = 1), are undefined.

The arithmetic instructions produce intermediate results which may be regarded as being infinitely precise. After normalization or denormalization, if the infinitely precise intermediate result is not representable in the destination format (either 32-bit or 64-bit) then it is rounded. The final result is then placed into the FPR in the double format.

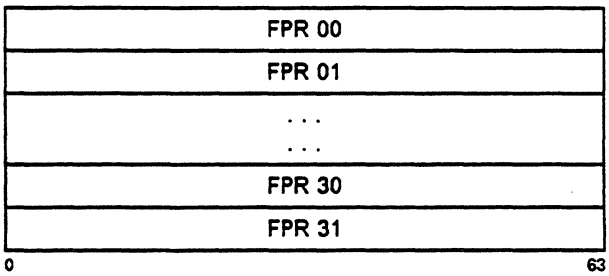


Figure 23. Floating-Point Registers

4.2.2 Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) controls the handling of floating-point exceptions and records status resulting from the floating-point operations. Bits 0:23 are status bits. Bits 24:31 are control bits.

The exception bits in the FPSCR (bits 0:12, 21:23) are sticky, with the exception of Floating-Point Enabled Exception Summary (FEX) and Floating-Point Invalid Operation Exception Summary (VX). That is, once set they remain set until they are cleared by an ***mcrfs***, ***mtfsfi***, ***mtfsf***, or ***mtfsb0*** instruction.

FEX and VX are simply the ORs of other FPSCR bits. Therefore these two bits are not listed among the FPSCR bits affected by the various instructions.

Figure 24. Floating-Point Status and Control Register

The format of the FPSCR is:

Bit(s) Description

0 ***Floating-Point Exception Summary*** (FX)
Every floating-point instruction shall implicitly set $FPSCR_{FX}$ if that instruction causes any of the floating-point exception bits in the FPSCR to transition from 0 to 1. ***mcrfs*** shall implicitly reset $FPSCR_{FX}$ if the FPSCR field containing $FPSCR_{FX}$ is copied. ***mtfsf***, ***mtfsfi***, ***mtfsb0***, and ***mtfsb1*** shall be able to set or clear $FPSCR_{FX}$ explicitly.

1 ***Floating-Point Enabled Exception Summary*** (FEX)
This bit signals the occurrence of any of the enabled exception conditions. It is the OR of all the floating-point exceptions masked with their respective enables. ***mcrfs*** shall implicitly reset $FPSCR_{FEX}$ if the result of the logical operation described above becomes zero. ***mtfsf***, ***mtfsfi***, ***mtfsb0***, and ***mtfsb1*** cannot set or clear $FPSCR_{FEX}$ explicitly.

2 ***Floating-Point Invalid Operation Exception Summary*** (VX)
This bit signals the occurrence of any invalid operation exception. It is the OR of all the Invalid Operation exceptions. ***mcrfs*** shall implicitly reset $FPSCR_{VX}$ if the result of the logical operation described above becomes zero. ***mtfsf***, ***mtfsfi***, ***mtfsb0***, and ***mtfsb1*** cannot set or clear $FPSCR_{VX}$ explicitly.

3 ***Floating-Point Overflow Exception*** (OX)
See Section 4.4.3, "Overflow Exception" on page 95.

4 ***Floating-Point Underflow Exception*** (UX)
See Section 4.4.4, "Underflow Exception" on page 95.

5 ***Floating-Point Zero Divide Exception*** (ZX)
See Section 4.4.2, "Zero Divide Exception" on page 94.

6 ***Floating-Point Inexact Exception*** (XX)
See Section 4.4.5, "Inexact Exception" on page 96.

7 ***Floating-Point Invalid Operation Exception (SNaN)*** (VXSNAN)
See Section 4.4.1, "Invalid Operation Exception" on page 93.

8 ***Floating-Point Invalid Operation Exception (∞—∞)*** (VXISI)
See Section 4.4.1, "Invalid Operation Exception" on page 93.

9 ***Floating-Point Invalid Operation Exception (∞÷∞)*** (VXIDI)
See Section 4.4.1, "Invalid Operation Exception" on page 93.

10 ***Floating-Point Invalid Operation Exception (0÷0)*** (VXZDZ)
See Section 4.4.1, "Invalid Operation Exception" on page 93.

11 ***Floating-Point Invalid Operation Exception (∞×0)*** (VXIMZ)
See Section 4.4.1, "Invalid Operation Exception" on page 93.

12 ***Floating-Point Invalid Operation Exception (Invalid Compare)*** (VXVC)
See Section 4.4.1, "Invalid Operation Exception" on page 93.

13 ***Floating-Point Fraction Rounded*** (FR)
The last floating-point instruction that potentially rounded the intermediate result incremented the fraction (see Section 4.3.6, "Rounding" on page 90). This bit is not sticky.

14 ***Floating-Point Fraction Inexact*** (FI)
The last floating-point instruction that potentially rounded the intermediate result produced an inexact fraction or a disabled exponent overflow (see Section 4.3.6, "Rounding" on page 90). This bit is not sticky.

15:19 ***Floating-Point Result Flags*** (FPRF)
This field is set as described below. For floating-point instructions other than the *Compare* instructions, the field is set based on the result placed into the target register, except that if any portion of the result is undefined then the value placed into the FPRF is undefined.

15 ***Floating-Point Result Class Descripter*** (C)
Floating-point instructions other than the *Compare* instructions may set this bit with the FPCC bits, to indicate the class of the result as shown in Figure 25 on page 86.

16:19 ***Floating-Point Condition Code*** (FPCC)
Floating-point *Compare* instructions set one of the FPCC bits to one and the other three FPCC bits to zero. Other floating-point instructions may set the FPCC bits with the C bit, to indicate the class of the result as shown in Figure 25 on page 86. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.

16 ***Floating-Point Less Than or Negative*** (FL or <)

17 ***Floating-Point Greater Than or Positive*** (FG or >)

18 ***Floating-Point Equal or Zero*** (FE or =)

19 ***Floating-Point Unordered or NaN*** (FU or ?)

20 Reserved

21 ***Floating-Point Invalid Operation Exception (Software Request)*** (VXSOFT)
This bit can be altered only by ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, or ***mtfsb1***. See Section 4.4.1, "Invalid Operation Exception" on page 93.

22 ***Floating-Point Invalid Operation Exception (Invalid Square Root)*** (VXSQRT)
See Section 4.4.1, "Invalid Operation Exception" on page 93.

Architecture Note

This bit is defined even for implementations that do not support either of the two optional instructions that set it, namely *Floating Square Root* and *Floating Reciprocal Square Root Estimate*. Defining it for all implementations gives software a standard interface for handling square root exceptions.

Programming Note

If the implementation does not support the *Floating Square Root* instruction or the *Floating Reciprocal Square Root Estimate* instruction, software can simulate the instruction and set this bit to reflect the exception.

23 ***Floating-Point Invalid Operation Exception (Invalid Integer Convert)*** (VXCVI)
See Section 4.4.1, "Invalid Operation Exception" on page 93.

24 ***Floating-Point Invalid Operation Exception Enable*** (VE)
See Section 4.4.1, "Invalid Operation Exception" on page 93.

25 ***Floating-Point Overflow Exception Enable***.(OE)
See Section 4.4.3, "Overflow Exception" on page 95.

26 ***Floating-Point Underflow Exception Enable*** (UE)
See Section 4.4.4, "Underflow Exception" on page 95.

27 ***Floating-Point Zero Divide Exception Enable*** (ZE)
See Section 4.4.2, "Zero Divide Exception" on page 94.

28 ***Floating-Point Inexact Exception Enable*** (XE)
See Section 4.4.5, "Inexact Exception" on page 96.

29 ***Floating-Point Non-IEEE Mode*** (NI)
If this bit is set to 1, the processor need not produce IEEE-conforming results for floating-point instructions, and the remaining FPSCR bits may have meanings other than those shown in this document. The operation of the processor when NI=1 is described in Book IV, *PowerPC Implementation Features* for the implementation, and may differ between implementations.

30:31 ***Floating-Point Rounding Control*** (RN)
See Section 4.3.6, "Rounding" on page 90.

00 Round to Nearest
01 Round toward Zero
10 Round toward +Infinity
11 Round toward –Infinity

Result Flags					Result Value Class
C	<	>	=	?	
1	0	0	0	1	Quiet NaN
0	1	0	0	1	– Infinity
0	1	0	0	0	– Normalized Number
1	1	0	0	0	– Denormalized Number
1	0	0	1	0	– Zero
0	0	0	1	0	+ Zero
1	0	1	0	0	+ Denormalized Number
0	0	1	0	0	+ Normalized Number
0	0	1	0	1	+ Infinity

Figure 25. Floating-Point Result Flags

Architecture Note

Setting Floating-Point Non-IEEE Mode (NI) to 1 is intended to permit results to be approximate, and to cause performance to be more predictable and less data-dependent than when NI=0. For example, in Non-IEEE Mode an implementation might return zero instead of a denormalized result, and a large number instead of an infinity. In Non-IEEE Mode an implementation should provide a means for ensuring that all results are produced without software assistance (i.e., without causing a Floating-Point Enabled type Program interrupt, a Floating-Point Assist interrupt, or a "fast trap": see Book III, *PowerPC Operating Environment Architecture*). The means may be controlled by one or more other FPSCR bits (recall that the other FPSCR bits have implementation-dependent meanings when NI=1).

4.3 Floating-Point Data

4.3.1 Data Format

This architecture defines the representation of a floating-point value in two different binary fixed length formats. The format may be a 32-bit single format for a single-precision value or a 64-bit double format for a double-precision value. The single format may be used for data in storage. The double format format may be used for data in storage and for data in floating-point registers.

The length of the exponent and the fraction fields differ between these two formats. The structure of the single and double formats is shown below:

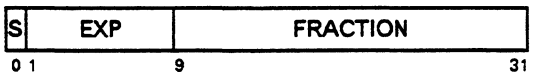


Figure 26. Floating-Point Single Format

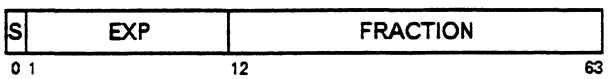


Figure 27. Floating-Point Double Format

Values in floating-point format are composed of three fields:

S	sign bit
EXP	exponent + bias
FRACTION	fraction

If only a portion of a floating-point data item in storage is accessed, such as with a load or store instruction for a byte or halfword (or word in the case of floating-point double format), the value affected will depend on whether the PowerPC system is operating with Big-Endian byte order (the default), or Little-Endian byte order. See Appendix D, "Little-Endian Byte Ordering" on page 145.

Representation of numerical values in the floating-point formats consist of a sign bit S, a biased exponent EXP, and the fraction portion FRACTION of the significand. The significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is a one for normalized numbers and a zero for denormalized numbers and is located in the unit bit position (i.e. the first bit to the left of the binary point). Values representable within the two floating-point formats can be specified by the parameters listed in Figure 28.

	Format	
	Single	**Double**
Exponent Bias	+127	+1023
Maximum Exponent	+127	+1023
Minimum Exponent	−126	−1022
Widths (bits)		
Format	32	64
Sign	1	1
Exponent	8	11
Fraction	23	52
Significand	24	53

Figure 28. IEEE Floating-Point Fields

The architecture requires that the FPRs of the Floating-Point Processor support the arithmetic instructions on values in the floating-point double format only.

4.3.2 Value Representation

This architecture defines numerical and non-numerical values representable within each of the two supported formats. The numerical values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The non-numerical values representable are the infinities, and the Not a Numbers (NaNs). The infinities are adjoined to the real numbers, but are not numbers themselves, and the standard rules of arithmetic do not hold when they appear in an operation. They are related to the reals by order alone. It is possible however to define restricted operations among numbers and infinities as defined below. The relative location on the real number line for each of the defined entities is shown in Figure 29 on page 88.

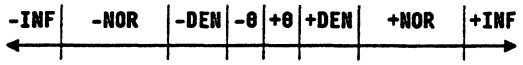


Figure 29. Approximation to Real Numbers

The NaNs are not related to the numbers or infinities by order or value but are encodings used to convey diagnostic information such as the representation of uninitialized variables.

The following is a description of the different floating-point values defined in the architecture:

Binary floating-point numbers
Machine representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.

Normalized numbers (±NOR)
These are values which have a biased exponent value in the range:

1 to 254 in single format
1 to 2046 in double format

They are values in which the implied unit bit is one. Normalized numbers are interpreted as follows:

$$NOR = (-1)^s \times 2^E \times (1.fraction)$$

where (s) is the sign, (E) is the unbiased exponent and (1.fraction) is the significand which is composed of a leading unit bit (implied bit) and a fraction part.

The ranges covered by the magnitude (M) of a normalized floating-point number are approximately equal to:

Single Format:

$$1.2 \times 10^{-38} \leq M \leq 3.4 \times 10^{38}$$

Double Format:

$$2.2 \times 10^{-308} \leq M \leq 1.8 \times 10^{308}$$

Zero values (±0)
These are values which have a biased exponent value of zero and a fraction value of zero. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (i.e., comparison regards +0 as equal to −0).

Denormalized numbers (±DEN)
These are values which have a biased exponent value of zero and a non-zero fraction value. They are non-zero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is zero. Denormalized numbers are interpreted as follows:

$$DEN = (-1)^s \times 2^{Emin} \times (0.fraction)$$

where Emin is the minimum representable exponent value (−126 for single-precision, −1022 for double-precision).

Infinities (±∞)
These are values which have the maximum biased exponent value:

255 in the single format
2047 in the double format

and a zero fraction value. They are used to approximate values greater in magnitude than the maximum normalized value.

Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the reals can be related by ordering in the affine sense:

$$-\infty < \text{every finite number} < +\infty$$

Arithmetic on infinities is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in Section 4.4.1, "Invalid Operation Exception" on page 93.

Not a Numbers (NaNs)
These are values which have the maximum biased exponent value and a non-zero fraction value. The sign bit is ignored (i.e. NaNs are neither positive nor negative). If the high-order bit of the fraction field is a zero then the NaN is a *Signalling NaN*, otherwise it is a *Quiet NaN*.

Signalling NaNs are used to signal exceptions when they appear as arithmetic operands.

Quiet NaNs are used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when Invalid Operation Exception is disabled ($FPSCR_{VE} = 0$). Quiet NaNs propagate through all operations except ordered comparison, *Floating Round to Single-Precision*, and conversion to integer. Quiet NaNs do not signal exceptions, except for ordered comparison and conversion to integer operations. Specific encodings, in QNaNs, can thus be preserved through a sequence of operations, and used to convey diagnostic information to help identify results from invalid operations.

When a QNaN is the result of an operation because one of the operands is a NaN or because a QNaN was generated due to a disabled Invalid Operation Exception, then the following rule is applied to determine the NaN with the high-order fraction bit set to one that is to be stored as the result.

if (FRA) is a NaN
  then FRT ← (FRA)
  else if (FRB) is a NaN

then if instruction is ***frsp***
then FRT ← $(FRB)_{0:34} \| {}^{29}0$
else FRT ← (FRB)
else if (FRC) is a NaN
then FRT ← (FRC)
else if generated QNaN
then FRT ← generated QNaN

If the operand specified by FRA is a NaN, then that NaN is stored as the result. Otherwise, if the operand specified by FRB is a NaN (if the instruction specifies an FRB operand), then that NaN is stored as the result, with the low-order 29 bits of the result set to 0 if the instruction is ***frsp***. Otherwise, if the operand specified by FRC is a NaN (if the instruction specifies an FRC operand), then that NaN is stored as the result. Otherwise, if a QNaN was generated due to a disabled Invalid Operation Exception, then that QNaN is stored as the result. If a QNaN is to be generated as a result, then the QNaN generated has a sign bit of zero, an exponent field of all ones, and a high-order fraction bit of one with all other fraction bits zero. Any instruction that generates a QNaN as the result of a disabled Invalid Operation must generate this QNaN (i.e., 0x7FF8_0000_0000_0000).

A double-precision NaN is considered to be representable in single format if and only if the low-order 29 bits of the double-precision NaN's fraction are zero.

4.3.3 Sign of Result

The following rules govern the sign of the result of an arithmetic operation, when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an addition operation is the sign of the input having the larger absolute value. The sign of the result of the subtraction operation x−y is the same as the sign of the result of the addition operation x+(−y).

  When the sum of two operands with opposite sign, or the difference of two operands with the same sign, is exactly zero, the sign of the result is positive in all rounding modes except Round toward −Infinity, in which mode the sign is negative.

- The sign of the result of a multiplication or division operation is the Exclusive OR of the signs of the inputs.

- The sign of the result of a *Square Root* or *Reciprocal Square Root Estimate* operation is always positive, except that the square root of −0 is −0 and the reciprocal square root of −0 is −Infinity.

- The sign of the result of a *Round to Single-Precision* or *Convert to/from Integer* operation is the sign of the input.

For the *Multiply-Add* instructions, the rules given above are applied first to the multiplication operation and then to the addition or subtraction operation (one of the inputs to the addition or subtraction operation is the result of the multiplication operation).

4.3.4 Normalization and Denormalization

When an arithmetic operation produces an intermediate result, consisting of a sign bit, an exponent, and a nonzero significand with a zero leading bit, it is not a normalized number and must be normalized before it is stored.

A number is normalized by shifting its significand left while decrementing its exponent by one for each bit shifted, until the leading significand bit becomes one. The guard bit and the round bit (see Section 4.5.1, "Execution Model for IEEE Operations" on page 96) participate in the shift with zeros shifted into the round bit. The exponent is regarded as if its range were unlimited. If the resulting exponent value is less than the minimum value that can be represented in the format specified for the result, the intermediate result is said to be "Tiny" and the stored result is determined by the rules described in Section 4.4.4, "Underflow Exception" on page 95. The sign of the number does not change.

When an arithmetic operation produces a non-zero intermediate result with an exponent value less than the minimum value that can be represented in the format specified for the result, the stored result is determined by the rules described in Section 4.4.4, "Underflow Exception" on page 95. This process may require denormalization.

A number is denormalized by shifting its significand right while incrementing its exponent by one for each bit shifted, until the exponent is equal to the format's minimum value. If any significant bits are lost in this shifting process then "Loss of Accuracy" has occurred (See Section 4.4.4, "Underflow Exception" on page 95) and Underflow Exception is signalled. The sign of the number does not change.

Engineering Note

When denormalized numbers are operands of multiply, divide, and square root operations, some implementations may prenormalize the operands internally before performing the operations.

4.3.5 Data Handling and Precision

Instructions are defined to move floating-point data between the FPRs and storage. For double format data the data is not altered during the move. For single format data, a format conversion from single to double is performed when loading from storage into an FPR and a format conversion from double to single is performed when storing from an FPR to storage. No floating-point exceptions are raised during these operations.

All arithmetic operations are performed using floating-point double format.

Floating-point single-precision is obtained with the implementation of four types of instruction.

1. Load Floating-Point Single

   This form of instruction accesses a single-precision operand in single format in storage, converts it to double-precision, and loads it into an FPR. ***No exceptions are detected on the load operation.***

2. Round to Floating-Point Single-Precision

   The *Floating Round to Single-Precision* instruction rounds a double-precision operand to single-precision if the operand is not already in single-precision range, checking the exponent for single-precision range and handling any exceptions according to respective enable bits, and places that operand into an FPR as a double-precision operand. For results produced by single-precision arithmetic instructions and by single-precision loads, this operation does not alter the value.

3. Single-Precision Arithmetic Instructions

   This form of instruction takes operands from the FPRs in double format, performs the operation as if it produced an intermediate result correct to infinite precision and with unbounded range, and then coerces this intermediate result to fit in single format. Status bits, in the FPSCR and in the Condition Register, are set to reflect the single-precision result. The result is then converted to double format and placed into an FPR. The result lies in the range supported by the single format.

   All input values must be representable in single format: if they are not, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the Condition Register (if Rc=1), are undefined.

4. Store Floating-Point Single

   This form of instruction converts a double-precision operand to single format and stores that operand into storage. ***No exceptions are detected on the store operation*** (the value being stored is effectively assumed to be the result of an instruction of one of the preceding three types).

When the result of a *Load Floating-Point Single*, *Floating Round to Single-Precision*, or single-precision arithmetic instruction is stored in an FPR, the low-order 29 FRACTION bits are zero.

Programming Note

The *Floating Round to Single-Precision* instruction is provided to allow value conversion from double-precision to single-precision with appropriate exception checking and rounding. This instruction should be used to convert double-precision floating-point values (produced by double-precision load and arithmetic instructions) to single-precision values prior to storing them into single format storage elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions can be stored directly, or used directly as operands for single-precision arithmetic instructions, without preceding the store, or the arithmetic instruction, by a *Floating Round to Single-Precision* instruction.

Programming Note

A single-precision value can be used in double-precision arithmetic operations. The reverse is not necessarily true (it is true only if the double-precision value is representable in single format).

Some implementations may execute single-precision arithmetic instructions faster than double-precision arithmetic instructions. Therefore, if double-precision accuracy is not required, single-precision data and instructions should be used.

4.3.6 Rounding

With the exception of the two optional *Estimate* instructions, *Floating Reciprocal Estimate Single* and *Floating Reciprocal Square Root Estimate*, all arithmetic instructions defined by this architecture produce an intermediate result that can be regarded as being infinitely precise. This result must then be written with a precision of finite length into an FPR. After normalization or denormalization, if the infinitely precise intermediate result is not representable in the precision required by the instruction then it is rounded before being placed into the target FPR.

The instructions that potentially round their result are the *Arithmetic, Multiply-Add*, and *Rounding and Conversion* instructions. For a given instance of one of these instructions, whether rounding occurs depends on the values of the inputs. Each of these instructions

sets FPSCR bits FR and FI, according to whether rounding occurred (FI) and whether the fraction was incremented (FR). If rounding occurred, FI is set to one, and FR may be set to either zero or one. If rounding did not occur, both FR and FI are set to zero.

The two *Estimate* instructions set FR and FI to undefined values. The remaining Floating-Point instructions do not alter FR and FI.

Four modes of rounding are provided which are user-selectable through the Floating-Point Rounding Control field in the FPSCR. See Section 4.2.2, "Floating-Point Status and Control Register" on page 85. These are encoded as follows:

RN	Rounding Mode
00	Round to Nearest
01	Round toward Zero
10	Round toward +Infinity
11	Round toward −Infinity

Let Z be the infinitely precise intermediate arithmetic result or the operand of a convert operation. If Z can be represented exactly in the target format, then no rounding occurs, and the result in all rounding modes is equivalent to truncation of Z. If Z cannot be represented exactly in the target format, let Z1 and Z2 be the next larger and next smaller numbers representable in the target format that bound Z, then Z1 or Z2 can be used to approximate the result in the target format.

Figure 30 shows the relation of Z, Z1, and Z2 in this case. The following rules specify the rounding in the four modes. "LSB" means "least significant bit."

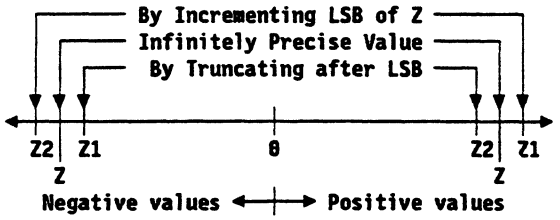


Figure 30. Selection of Z1 and Z2

Round to Nearest
Choose the best approximation of Z1 or Z2. In case of a tie, choose the one which is even (least significant bit 0).

Round toward Zero
Choose the smaller in magnitude (Z1 or Z2).

Round toward +Infinity
Choose Z1.

Round toward −Infinity
Choose Z2.

See Section 4.5.1, "Execution Model for IEEE Operations" on page 96 for a detailed explanation of rounding.

If Z is to be rounded up and Z1 does not exist (i.e., if there is no number larger than Z that is representable in the target format), then an Overflow Exception occurs if Z is positive and an Underflow Exception occurs if Z is negative. Similarly, if Z is to be rounded down and Z2 does not exist, then an Overflow Exception occurs if Z is negative and an Underflow Exception occurs if Z is positive. The results in these cases are defined in Section 4.4, "Floating-Point Exceptions" on page 91.

4.4 Floating-Point Exceptions

This architecture defines the following floating-point exceptions:

- Invalid Operation Exception
  - SNaN
  - Infinity−Infinity
  - Infinity÷Infinity
  - Zero÷Zero
  - Infinity×Zero
  - Invalid Compare
  - Software Request
  - Invalid Square Root
  - Invalid Integer Convert
- Zero Divide Exception
- Overflow Exception
- Underflow Exception
- Inexact Exception

These exceptions may occur during execution of floating-point arithmetic instructions. In addition, an Invalid Operation Exception occurs when a *Status and Control Register* instruction sets $FPSCR_{VXSOFT}$ to 1 (Software Request). An Invalid Square Root operation can occur only if at least one of the *Floating Square Root* instructions defined in Appendix A, "Optional Instructions" on page 119, is implemented.

Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. The exception bit indicates occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with MSR bits FE0 and FE1, whether and how the system floating-point enabled exception error handler is invoked. The MSR (Machine State Register) is described in Book III, *PowerPC Operating Environment Architecture*. (In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurence of an exception depends only on the instruction and its inputs, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow Exception may depend on the setting of the enable bit.)

The Floating-Point Exception Summary bit (FX) in the FPSCR is set when any of the exception bits transitions from a zero to a one or when explicitly set by software. The Floating-Point Enabled Exception Summary bit (FEX) in the FPSCR is set when any of the exceptions is set and the exception is enabled (enable bit is one).

A single instruction, other than ***mtfsfi*** or ***mtfsf***, may set more than one exception in the following cases:

- Inexact Exception may be set with Overflow Exception.
- Inexact Exception may be set with Underflow Exception.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception ($\infty \times 0$) for *Multiply-Add* instructions.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Compare) for *Compare Ordered* instructions.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Integer Convert) for *Convert to Integer* instructions.

When an exception occurs the instruction execution may be suppressed or a result may be delivered, depending on the exception.

Instruction execution is suppressed for the following kinds of exception, so that there is no possibility that one of the operands is lost.

- Enabled Invalid Operation
- Enabled Zero Divide

For the remaining kinds of exception, a result is generated and written to the destination specified by the instruction causing the exception. The result may be a different value for the enabled and disabled conditions for some of these exceptions. The kinds of exception that deliver a result are the following.

- Disabled Invalid Operation
- Disabled Zero Divide
- Disabled Overflow
- Disabled Underflow
- Disabled Inexact
- Enabled Overflow
- Enabled Underflow
- Enabled Inexact

Subsequent sections define each of the floating-point exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of "traps" and "trap handlers." In this architecture, an FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the "trap enabled" case: the expectation is that the exception will be detected by software, which will revise the result. An FPSCR exception enable bit of 0 causes generation of the "default result" value specified for the "trap disabled" (or "no trap occurs" or "trap is not implemented") case: the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the sections below.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is desired for all exceptions, all FPSCR exception enable bits should be set to 0 and Ignore Exceptions Mode (see below) should be used. In this case the system floating-point enabled exception error handler is not invoked, even if floating-point exceptions occur: software can inspect the FPSCR exception bits if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to 1 and a mode other than Ignore Exceptions Mode must be used. In this case the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs.

Whether and how the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs is controlled by MSR bits FE0 and FE1, as follows. (The system floating-point enabled exception error handler is never invoked because of a disabled floating-point exception.)

FE0 FE1 Description

0 0 Ignore Exceptions Mode
Floating-point exceptions do not cause the system floating-point enabled exception error handler to be invoked.

0 1 Imprecise Nonrecoverable Mode
The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. The state of the processor may include conditions and data affected by the exception (i.e., hazards are not avoided). It may not be possible to identify the excepting instruction nor the data that caused the exception (i.e., the data is not recoverable).

1 0 Imprecise Recoverable Mode
The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. Sufficient information is provided to the system floating-point enabled exception error handler that it can identify the excepting instruction and the operands, and correct the result. All

hazards caused by the exception are avoided (e.g., use of the data that would have been produced by the excepting instruction).

1 1 **Precise Mode**
The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.

In all cases the question of whether or not a floating-point result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by any MSR bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system floating-point enabled exception error handler is invoked have completed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has been executed. (Recall that, for the two Imprecise modes, the instruction at which the system floating-point enabled exception error handler is invoked need not be the instruction that caused the exception.) The instruction at which the system floating-point enabled exception error handler is invoked has not been executed, unless it is the excepting instruction, in which case it has been executed unless the kind of exception is among those listed above as suppressed.

Programming Note

In any of the three non-Precise modes, a *Floating-Point Status and Control Register* instruction can be used to force any exceptions, due to instructions initiated before the *Floating-Point Status and Control Register* instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In either of the Imprecise modes, a *Floating-Point Status and Control Register* instruction can be used to force any invocations of the system floating-point enabled exception error handler, due to instructions initiated before the *Floating-Point Status and Control Register* instruction, to occur. (This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.)

A ***sync*** instruction also has the effects described above, but is likely to degrade performance more than a *Floating-Point Status and Control Register* instruction.

In order to obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

- If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used, with all FPSCR exception enable bits set to 0.
- If the IEEE default results are not acceptable to the application, Imprecise Non-Recoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to 1 for those exceptions for which the system floating-point enabled exception error handler is to be invoked.
- Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to 1.
- Precise Mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

Engineering Note

It is permissible for the implementation to be precise in any of the three modes that permit exceptions, or to be recoverable in Non-Recoverable Mode.

4.4.1 Invalid Operation Exception

4.4.1.1 Definition

An Invalid Operation Exception occurs whenever an operand is invalid for the specified operation. The invalid operations are:

- Any operation, except *Load, Store, Move, Select,* and ***mtfsf***, on a signalling NaN (SNaN)
- For add or subtract operations, magnitude subtraction of infinities ($\infty - \infty$)
- Division of infinity by infinity ($\infty \div \infty$)
- Division of zero by zero ($0 \div 0$)
- Multiplication of infinity by zero ($\infty \times 0$)
- Ordered comparison involving a NaN (Invalid Compare)
- Square root or reciprocal square root of a negative (and nonzero) number (Invalid Square Root)
- Integer convert involving a large number, an infinity, or a NaN (Invalid Integer Convert)

In addition, an Invalid Operation Exception occurs if software explicitly requests this by executing a ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction that sets $FPSCR_{VXSOFT}$ to 1 (Software Request). An Invalid Square Root operation can occur only if at least one of the *Floating Square Root* instructions defined in Appendix A, "Optional Instructions" on page 119, is implemented.

Programming Note

The purpose of $FPSCR_{VXSOFT}$ is to allow software to cause an Invalid Operation Exception for a condition that is not necessarily associated with the execution of a floating-point instruction. For example, it might be set by a program that computes a square root, if the source operand is negative.

4.4.1.2 Action

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

When Invalid Operation Exception is enabled ($FPSCR_{VE}=1$) and Invalid Operation occurs or software explicitly requests the exception then the following actions are taken:

1. One or two Invalid Operation Exceptions is set
   - $FPSCR_{VXSNAN}$ (if SNaN)
   - $FPSCR_{VXISI}$ (if $\infty-\infty$)
   - $FPSCR_{VXIDI}$ (if $\infty\div\infty$)
   - $FPSCR_{VXZDZ}$ (if $0\div0$)
   - $FPSCR_{VXIMZ}$ (if $\infty\times0$)
   - $FPSCR_{VXVC}$ (if invalid comp)
   - $FPSCR_{VXSOFT}$ (if software req)
   - $FPSCR_{VXSQRT}$ (if invalid sqrt)
   - $FPSCR_{VXCVI}$ (if invalid int cvrt)
2. If the operation is an arithmetic, *Floating Round to Single-Precision*, or convert to integer operation,
   - the target FPR is unchanged
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is unchanged
3. If the operation is a compare,
   - $FPSCR_{FR\ FI\ C}$ are unchanged
   - $FPSCR_{FPCC}$ is set to reflect unordered
4. If software explicitly requests the exception,
   - $FPSCR_{FR\ FI\ FPRF}$ are as set by the ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction

When Invalid Operation Exception is disabled ($FPSCR_{VE}=0$) and Invalid Operation occurs or software explicitly requests the exception then the following actions are taken:

1. One or two Invalid Operation Exceptions is set
   - $FPSCR_{VXSNAN}$ (if SNaN)
   - $FPSCR_{VXISI}$ (if $\infty-\infty$)
   - $FPSCR_{VXIDI}$ (if $\infty\div\infty$)
   - $FPSCR_{VXZDZ}$ (if $0\div0$)
   - $FPSCR_{VXIMZ}$ (if $\infty\times0$)
   - $FPSCR_{VXVC}$ (if invalid comp)
   - $FPSCR_{VXSOFT}$ (if software req)
   - $FPSCR_{VXSQRT}$ (if invalid sqrt)
   - $FPSCR_{VXCVI}$ (if invalid int cvrt)
2. If the operation is an arithmetic or *Floating Round to Single-Precision* operation
   - the target FPR is set to a Quiet NaN
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is set to indicate the class of the result (Quiet NaN)
3. If the operation is a convert to 32-bit integer operation,
   - the target FPR is set as follows:
     - $FRT_{0:31} \leftarrow$ undefined
     - $FRT_{32:63} \leftarrow$ most negative 32-bit integer
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is undefined
4. If the operation is a convert to 64-bit integer operation,
   - the target FPR is set as follows:
     - $FRT_{0:63} \leftarrow$ most negative 64-bit integer
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is undefined
5. If the operation is a compare,
   - $FPSCR_{FR\ FI\ C}$ are unchanged
   - $FPSCR_{FPCC}$ is set to reflect unordered
6. If software explicitly requests the exception,
   - $FPSCR_{FR\ FI\ FPRF}$ are as set by the ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction

4.4.2 Zero Divide Exception

4.4.2.1 Definition

A Zero Divide Exception occurs when a *Divide* instruction is executed with a zero divisor value and a finite non-zero dividend value. It also occurs when a *Reciprocal Square Root Estimate* instruction is executed with an operand value of zero.

Architecture Note

The name is a misnomer used for historical reasons. The proper name for this exception should be "Exact Infinite Result from Finite Operands" corresponding to what mathematicians call a "pole."

4.4.2.2 Action

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

When Zero Divide Exception is enabled ($FPSCR_{ZE}=1$) and Zero Divide occurs then the following actions are taken:

1. Zero Divide Exception is set
   - $FPSCR_{ZX} \leftarrow 1$
2. The target FPR is unchanged
3. $FPSCR_{FR\ FI}$ are set to zero
4. $FPSCR_{FPRF}$ is unchanged

When Zero Divide Exception is disabled ($FPSCR_{ZE}=0$) and Zero Divide occurs then the following actions are taken:

1. Zero Divide Exception is set
   - $FPSCR_{ZX} \leftarrow 1$

2. The target FPR is set to a ±Infinity, where the sign is determined by the XOR of the signs of the operands
3. $FPSCR_{FR\ FI}$ are set to zero
4. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (±Infinity)

4.4.3 Overflow Exception

4.4.3.1 Definition

Overflow occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

4.4.3.2 Action

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

When Overflow Exception is enabled ($FPSCR_{OE}=1$) and exponent overflow occurs then the following actions are taken:

1. Overflow Exception is set
   $FPSCR_{OX} \leftarrow 1$
2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by subtracting 1536
3. For single-precision arithmetic instructions and the *Floating Round to Single-Precision* instruction, the exponent of the normalized intermediate result is adjusted by subtracting 192
4. The adjusted rounded result is placed into the target FPR
5. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (±Normal Number)

When Overflow Exception is disabled ($FPSCR_{OE}=0$) and overflow occurs then the following actions are taken:

1. Overflow Exception is set
   $FPSCR_{OX} \leftarrow 1$
2. Inexact Exception is set
   $FPSCR_{XX} \leftarrow 1$
3. The result is determined by the rounding mode ($FPSCR_{RN}$) and the sign of the intermediate result as follows:
   A. Round to Nearest
      Store ± Infinity, where the sign is the sign of the intermediate result
   B. Round toward Zero
      Store the format's largest finite number with the sign of the intermediate result
   C. Round toward +Infinity
      For negative overflows, store the format's most negative finite number; for positive overflows, store +Infinity
   D. Round toward −Infinity
      For negative overflows, store −Infinity; for positive overflows, store the format's largest finite number
4. The result is placed into the target FPR
5. $FPSCR_{FR}$ is set to one if the result is incremented when rounded, and otherwise to zero
6. $FPSCR_{FI}$ is set to one
7. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (±Infinity or ±Normal Number)

4.4.4 Underflow Exception

4.4.4.1 Definition

Underflow Exception is defined separately for the enabled and disabled states:

- Enabled:
  Underflow occurs when the intermediate result is "Tiny."
- Disabled:
  Underflow occurs when the intermediate result is "Tiny" and there is "Loss of Accuracy."

  A "Tiny" result is detected before rounding, when a nonzero result value computed as though the exponent range were unbounded would be less in magnitude than the smallest normalized number.

  If the intermediate result is "Tiny" and the Underflow Exception Enable is off ($FPSCR_{UE}=0$) then the intermediate result is to be denormalized (Section 4.3.4, "Normalization and Denormalization" on page 89) and rounded (Section 4.3.6, "Rounding" on page 90).

  "Loss of Accuracy" is detected when the delivered result value differs from what would have been computed were both the exponent range and precision unbounded.

4.4.4.2 Action

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

When Underflow Exception is enabled ($FPSCR_{UE}=1$) and exponent underflow occurs then the following actions are taken:

1. Underflow Exception is set
   $FPSCR_{UX} \leftarrow 1$
2. For double-precision arithmetic and conversion instructions, the exponent of the normalized intermediate result is adjusted by adding 1536
3. For single-precision arithmetic instructions and the *Floating Round to Single-Precision* instruction, the exponent of the normalized intermediate result is adjusted by adding 192
4. The adjusted rounded result is placed into the target FPR
5. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (±Normalized Number)

Programming Note

The FR and FI bits are provided to allow the system floating-point enabled exception error handler, when invoked because of an Underflow Exception, to simulate a "trap disabled" environment. That is, the FR and FI bits allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized.

When Underflow Exception is disabled ($FPSCR_{UE} = 0$) and underflow occurs then the following actions are taken:

1. Underflow Exception is set
   $FPSCR_{UX} \leftarrow 1$
2. The rounded result is placed into the target FPR
3. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (±Denormalized Number or ±Zero)

4.4.5 Inexact Exception

4.4.5.1 Definition

Inexact Exception occurs when one of two conditions occur during rounding:

1. The rounded result differs from the intermediate result assuming the intermediate result exponent range and precision to be unbounded.
2. The rounded result overflows and Overflow Exception is disabled.

4.4.5.2 Action

The action to be taken does not depend on the setting of the Inexact Exception Enable bit of the FPSCR.

When Inexact Exception occurs then the following actions are taken:

1. Inexact Exception is set
   $FPSCR_{XX} \leftarrow 1$
2. The rounded or overflowed result is placed into the target FPR
3. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result

Programming Note

In some implementations, enabling Inexact Exceptions may degrade performance more than enabling other types of floating-point exception.

4.5 Floating-Point Execution Models

All implementations of this architecture must provide the equivalent of the following execution models to insure that identical results are obtained.

Special rules are provided in the definition of the arithmetic instructions for the infinities, denormalized numbers and NaNs.

Although the double format specifies an 11-bit exponent, exponent arithmetic makes use of two additional bit positions to avoid potential transient overflow conditions. One extra bit is required when denormalized double-precision numbers are prenormalized. The second bit is required to permit the computation of the adjusted exponent value in the following cases when the corresponding exception enable bits is one:

- Underflow during multiplication using a denormalized factor.
- Overflow during division using a denormalized divisor.

The IEEE standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision arithmetic instructions to have either (or both) single-precision or double-precision operands, but states that single-precision arithmetic instructions should not accept double-precision operands. The PowerPC Architecture follows these guidelines: double-precision arithmetic instructions can have operands of either or both precisions, while single-precision arithmetic instructions require all operands to be single-precision. Double-precision arithmetic instructions produce double-precision values, while single-precision arithmetic instructions produce single-precision values.

For arithmetic instructions, conversions from double-precision to single-precision must be done explicitly by software, while conversions from single-precision to double-precision are done implicitly.

4.5.1 Execution Model for IEEE Operations

The following description uses 64-bit arithmetic as an example. 32-bit arithmetic is similar except that the FRACTION is a 23-bit field, and the single-precision Guard, Round, and Sticky bits (described in this section) are logically adjacent to the 23-bit FRACTION field.

IEEE-conforming significand arithmetic is considered to be performed with a floating-point accumulator having the following format:

Figure 31. IEEE 64-bit Execution Model

The S bit is the sign bit.

The C bit is the carry bit that captures the carry out of the significand.

The L bit is the leading unit bit of the significand which receives the implicit bit from the operands.

The FRACTION is a 52-bit field which accepts the fraction of the operands.

The Guard (G), Round (R), and Sticky (X) bits are extensions to the low order bits of the accumulator. The G and R bits are required for post normalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits which may appear to the low-order side of the R bit, either due to shifting the accumulator right or other generation of low-order result bits. The G and R bits participate in the left shifts with zeros being shifted into the R bit. Figure 32 shows the significance of the G, R, and X bits with respect to the intermediate result (IR), the next lower in magnitude representable number (NL), and the next higher in magnitude representable number (NH).

G R X	Interpretation
0 0 0	IR is exact
0 0 1 0 1 0 0 1 1	IR closer to NL
1 0 0	IR midway between NL & NH
1 0 1 1 1 0 1 1 1	IR closer to NH

Figure 32. Interpretation of G, R, and X bits

The significand of the intermediate result is made up of the L bit, the FRACTION, and the G,R and X bits.

The infinitely precise intermediate result of an operation is the result normalized in bits L, FRACTION, G, R, and X of the floating-point accumulator.

Before the results are stored into an FPR, the significand is rounded if necessary, using the rounding mode specified by $FPSCR_{RN}$. If rounding results in a carry into C, the significand is shifted right one position and the exponent incremented by one. This, in turn, may yield an inexact result and possibly also exponent overflow. Fraction bits to the left of the bit position used for rounding are stored into the FPR and low-order bit positions, if any, are set to zero.

Four rounding modes are provided which are user-selectable through $FPSCR_{RN}$ as decribed in Section 4.3.6, "Rounding" on page 90. For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. Figure 33 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers.

Format	Guard	Round	Sticky
Double	G bit	R bit	X bit
Single	24	25	26:52 G,R,X

Figure 33. Location of the Guard, Round and Sticky Bits

Rounding can be treated as though the significand were shifted right, if required, until the least significant bit to be retained is in the low-order bit position of the FRACTION. If any of the Guard, Round, or Sticky bits are nonzero, then the result is inexact.

Z1 and Z2, as defined on page 91, can be used to approximate the result in the target format when one of the following rules is used.

- ***Round to Nearest***

  Guard bit = 0
  The result is truncated. (Result exact (GRX = 000) or closest to next lower value in magnitude (GRX = 001, 010, or 011))

  Guard bit = 1
  Depends on Round and Sticky bits:

  Case a
  If the Round or Sticky bit is one (inclusive), the result is incremented. (Result closest to next higher value in magitude (GRX = 101, 110, or 111))

  Case b
  If the Round and Sticky bits are zero (result midway between closest representable values) then if the low-order bit of the result is one the result is incremented. Otherwise (the low-order bit of the result is zero) the result is truncated (this is the case of a tie rounded to even).

  If during the Round to Nearest process, truncation of the unrounded number would produce the maximum magnitude for the specified precision, then the following action is taken:

Guard bit = 1
Store infinity with the sign of the unrounded result.

Guard bit = 0
Store the truncated (maximum magnitude) value.

- ***Round toward Zero***
  Choose the smaller in magnitude of Z1 or Z2. See "Rounding" on page 91 for the definitions of Z1 and Z2. If Guard, Round, or Sticky bit is nonzero, the result is inexact.
- ***Round toward +Infinity***
  Choose Z1. See "Rounding" on page 91 for the definition of Z1.
- ***Round toward −Infinity***
  Choose Z2. See "Rounding" on page 91 for the definition of Z2.

Where the result is to have fewer than 53 bits of precision because the instruction is a *Floating Round to Single-Precision* or single-precision arithmetic instruction, the intermediate result either is normalized or is placed in correct denormalized form before the result is potentially rounded.

4.5.2 Execution Model for Multiply-Add Type Instructions

The PowerPC Architecture makes use of a special form of instruction which performs up to three operations in one instruction (a multiply, an add and a negate). With this added capability is the special feature of being able to produce a more exact intermediate result as an input to the rounder. 32-bit arithmetic is similar except that the FRACTION field is smaller.

The multiply-add operations produce intermediate results conforming to the following model:

Figure 34. Multiply-Add Execution Model

The first part of the operation is a multiply. The multiply has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), then the significand is shifted right one position, shifting the L bit (leading unit bit) into the most significant bit of the fraction and shifting the C bit (carry out) into the L bit. All 106 bits (L bit, the fraction) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount which is added to that exponent to make it equal to the other input's exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X′ bit. The add operation also produces a result conforming to the above model with the X′ bit taking part in the add operation.

The result of the add is then normalized, with all bits of the add result, except the X′ bit, participating in the shift. The normalized result provides an intermediate result as input to the rounder which conforms to the model described in Section 4.5.1, "Execution Model for IEEE Operations" on page 96, where:

- The Guard bit is bit 53 of the intermediate result.
- The Round bit is bit 54 of the intermediate result.
- The Sticky bit is the OR of all remaining bits to the right of bit 55, inclusive.

The rules of rounding the intermediate result are the same as the described in Section 4.5.1, "Execution Model for IEEE Operations" on page 96.

If the instruction is *Floating Negative Multiply-Add* or *Floating Negative Multiply-Subtract* the final result is negated.

Status bits are set to reflect the result of the entire operation: e.g., no status is recorded for the result of the multiplication part of the operation.

4.6 Floating-Point Processor Instructions

Architecture Note

The rules followed in assigning new primary and extended opcodes, for instructions that are not in the Power Architecture, are the following.

1. A new primary opcode, 59, has been added. It is used for the single-precision arithmetic instructions.
2. The single-precision instructions for which there is a corresponding double-precision instruction have the same format and extended opcode as that double-precision instruction.
3. In assigning new extended opcodes for primary opcode 63, the following regularities, present in the Power Architecture, have been maintained. In addition, all new X-form instructions in primary opcode 63 have bits 21:22 = 0b11, which distinguishes them from the X-form instructions present in Power Architecture.
   - Bit 26 = 1 iff the instruction is A-form.
   - Bits 26:29 = 0b0000 iff the instruction is a comparison or ***mcrfs*** (i.e., iff the instruction sets an explicitly-designated CR field).
   - Bits 26:28 = 0b001 iff the instruction explicitly refers to or sets the FPSCR (i.e., is a *Floating-Point Status and Control Register* instruction) and is not ***mcrfs***.
   - Bits 26:30 = 0b01000 iff the instruction is a *Move Register* instruction, or any other instruction that does not refer to or set the FPSCR.
4. In assigning extended opcodes for primary opcode 59, the following regularities have been maintained. They are based on those rules for primary opcode 63 that apply to the instructions having primary opcode 59. In particular, primary opcode 59 has no *Floating-Point Status and Control Register* instructions, so the corresponding rule does not apply.
   - If there is a corresponding instruction with primary opcode 63, its extended opcode is used.
   - Bit 26 = 1 iff the instruction is A-form.
   - Bits 26:30 = 0b01000 iff the instruction is a *Move Register* instruction, or any other instruction that does not refer to or set the FPSCR.

4.6.1 Floating-Point Storage Access Instructions

The *Storage Access* instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.11.2, "Effective Address Calculation" on page 12.

The order of bytes accessed by floating-point loads and stores is Big-Endian, unless Little-Endian storage ordering is selected as described in Appendix D, "Little-Endian Byte Ordering" on page 145.

Programming Note

The "la" extended mnemonic permits computing an Effective Address as a *Load* or *Store* instruction would, but loads the address itself into a GPR rather than loading the value that is in storage at that address. This extended mnemonic is described in "Load Address" on page 144.

4.6.1.1 Storage Access Exceptions

Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (*Store* only), or if the program attempts to access storage that is unavailable to it.

When PowerPC is executing with Little-Endian byte ordering, the system alignment error handler will be invoked whenever a floating-point load or store instruction is executed that specifies an unaligned operand. See Appendix D, "Little-Endian Byte Ordering" on page 145.

4.6.2 Floating-Point Load Instructions

There are two basic forms of load instruction, single-precision and double-precision. Because the FPRs support only floating-point double format, single-precision *Load Floating-Point* instructions convert single-precision data to double format prior to loading the operands into the target FPR. The conversion and loading steps are as follows:

Let $WORD_{0:31}$ be the floating-point single-precision operand accessed from storage.

Normalized Operand

if $WORD_{1:8} > 0$ and $WORD_{1:8} < 255$ then
  $FRT_{0:1} \leftarrow WORD_{0:1}$
  $FRT_2 \leftarrow \neg WORD_1$
  $FRT_3 \leftarrow \neg WORD_1$
  $FRT_4 \leftarrow \neg WORD_1$
  $FRT_{5:63} \leftarrow WORD_{2:31} \| {}^{29}0$

Denormalized Operand

if $WORD_{1:8} = 0$ and $WORD_{9:31} \neq 0$ then
  sign $\leftarrow WORD_0$
  exp $\leftarrow -126$
  $frac_{0:52} \leftarrow 0b0 \| WORD_{9:31} \| {}^{29}0$
  normalize the operand
    Do while $frac_0 = 0$
      $frac \leftarrow frac_{1:52} \| 0b0$
      exp $\leftarrow$ exp $-$ 1
    End
  $FRT_0 \leftarrow$ sign
  $FRT_{1:11} \leftarrow$ exp + 1023
  $FRT_{12:63} \leftarrow frac_{1:52}$

Zero / Infinity / NaN

if $WORD_{1:8} = 255$ or $WORD_{1:31} = 0$ then
  $FRT_{0:1} \leftarrow WORD_{0:1}$
  $FRT_2 \leftarrow WORD_1$
  $FRT_3 \leftarrow WORD_1$
  $FRT_4 \leftarrow WORD_1$
  $FRT_{5:63} \leftarrow WORD_{2:31} \| {}^{29}0$

Engineering Note

The above description of the conversion steps is a model only. The actual implementation may vary from this but must produce results equivalent to what this model would produce.

For double-precision *Load Floating-Point* instructions, no conversion is required as the data from storage is copied directly into the FPR.

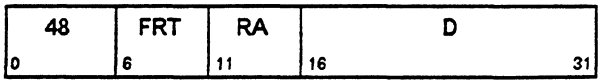
Many of the *Load Floating-Point* instructions have an "update" form, in which register RA is updated with the effective address. For these forms, if RA≠0, the effective address is placed into register RA and the storage element (word or doubleword) addressed by EA is loaded into FRT.

Note: Recall that RA, RB, and RT denote General Purpose Registers, while FRA, FRB, FRC and FRT denote Floating-Point Registers.

Byte order of PowerPC is Big-Endian by default; see Appendix D, "Little-Endian Byte Ordering" on page 145 for PowerPC systems operated with Little-Endian byte ordering.

Load Floating-Point Single D-form

lfs FRT,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
FRT ← DOUBLE(MEM(EA, 4))
```

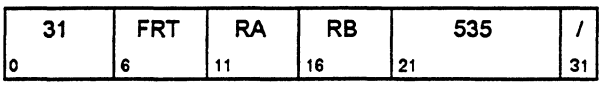
Let the effective address (EA) be the sum (RA|0)+D.

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 100) and placed into register FRT.

Special Registers Altered:
None

Load Floating-Point Single Indexed X-form

lfsx FRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
FRT ← DOUBLE(MEM(EA, 4))
```

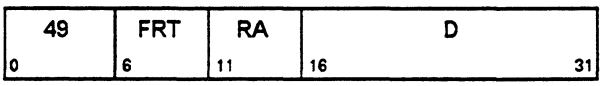
Let the effective address (EA) be the sum (RA|0)+(RB).

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 100) and placed into register FRT.

Special Registers Altered:
None

Load Floating-Point Single with Update D-form

lfsu FRT,D(RA)




```
EA ← (RA) + EXTS(D)
FRT ← DOUBLE(MEM(EA, 4))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D.

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 100) and placed into register FRT.

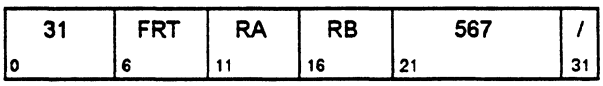
EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Load Floating-Point Single with Update Indexed X-form

lfsux FRT,RA,RB




```
EA ← (RA) + (RB)
FRT ← DOUBLE(MEM(EA, 4))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB).

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 100) and placed into register FRT.

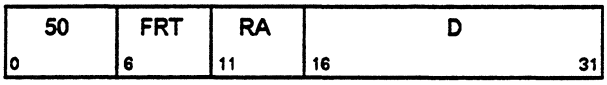
EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Load Floating-Point Double D-form

lfd FRT,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
FRT ← MEM(EA, 8)
```

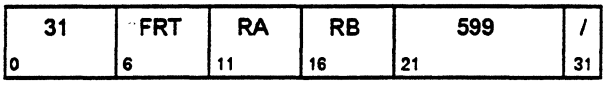
Let the effective address (EA) be the sum (RA|0)+D.

The doubleword in storage addressed by EA is placed into register FRT.

Special Registers Altered:
None

Load Floating-Point Double Indexed X-form

lfdx FRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
FRT ← MEM(EA, 8)
```

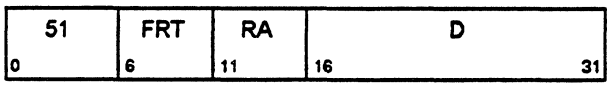
Let the effective address (EA) be the sum (RA|0)+(RB).

The doubleword in storage addressed by EA is placed into register FRT.

Special Registers Altered:
None

Load Floating-Point Double with Update D-form

lfdu FRT,D(RA)




```
EA ← (RA) + EXTS(D)
FRT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D.

The doubleword in storage addressed by EA is placed into register FRT.

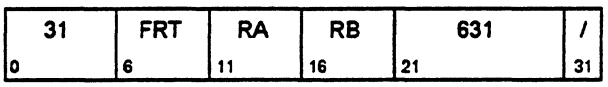
EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Load Floating-Point Double with Update Indexed X-form

lfdux FRT,RA,RB




```
EA ← (RA) + (RB)
FRT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB).

The doubleword in storage addressed by EA is placed into register FRT.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

4.6.3 Floating-Point Store Instructions

There are three basic forms of store instruction, single-precision, double-precision, and integer. The integer form is provided by the optional *Store Floating-Point as Integer Word* instruction, described on page 120. Because the FPRs support only floating-point double format for floating-point data, single-precision *Store Floating-Point* instructions convert double-precision data to single format prior to storing the operands into storage. The conversion steps are as follows:

Let $WORD_{0:31}$ be the word in storage written to.

No Denormalization Required (includes Zero / Infinity / NaN)

```
if FRS1:11 > 896 or FRS1:63 ≠ 0 then
    WORD0:1 ← FRS0:1
    WORD2:31 ← FRS5:34
```

Denormalization Required

```
if 874 ≤ FRS1:11 ≤ 896 then
    sign ← FRS0
    exp ← FRS1:11 − 1023
    frac ← 0b1 || FRS12:63
    Denormalize operand
        Do while exp < −126
            frac ← 0b0 || frac0:62
            exp ← exp + 1
        End
    WORD0 ← sign
    WORD1:8 ← 0x00
    WORD9:31 ← frac1:23
else WORD ← undefined
```

Notice that if the value to be stored by a single-precision *Store Floating-Point* instruction is larger in magnitude than the maximum number representable in single format, the first case above (No Denormalization Required) applies. The result stored in WORD is then a well-defined value, but is not numerically equal to the value in the source register (i.e., the result of a single-precision *Load Floating-Point* from WORD will not compare equal to the contents of the original source register).

Engineering Note

The above description of the conversion steps is a model only. The actual implementation may vary from this but must produce results equivalent to what this model would produce.

For double-precision *Store Floating-Point* instructions and for the *Store Floating-Point as Integer Word* instruction, no conversion is required as the data from the FPR is copied directly into storage.

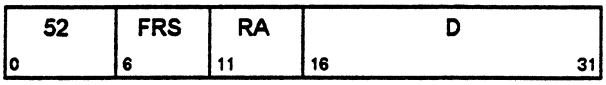
Many of the *Store Floating-Point* instructions have an "update" form, in which register RA is updated with the effective address. For these forms, if RA≠0, the effective address is placed into register RA.

Note: Recall that RA, RB, and RT denote General Purpose Registers, while FRA, FRB, FRC and FRT denote Floating-Point Registers.

Byte order of PowerPC is Big-Endian by default; see Appendix D, "Little-Endian Byte Ordering" on page 145 for PowerPC systems operated with Little-Endian byte ordering.

Store Floating-Point Single D-form

stfs FRS,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 4) ← SINGLE(FRS)
```

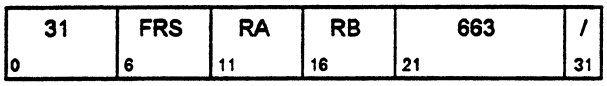
Let the effective address (EA) be the sum (RA|0)+D.

The contents of register FRS is converted to single format (see page 103) and stored into the word in storage addressed by EA.

Special Registers Altered:
None

Store Floating-Point Single Indexed X-form

stfsx FRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← SINGLE(FRS)
```

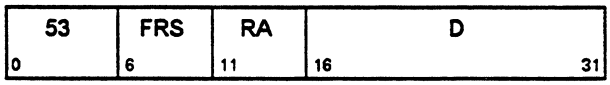
Let the effective address (EA) be the sum (RA|0)+(RB).

The contents of register FRS is converted to single format (see page 103) and stored into the word in storage addressed by EA.

Special Registers Altered:
None

Store Floating-Point Single with Update D-form

stfsu FRS,D(RA)




```
EA ← (RA) + EXTS(D)
MEM(EA, 4) ← SINGLE(FRS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D.

The contents of register FRS is converted to single format (see page 103) and stored into the word in storage addressed by EA.

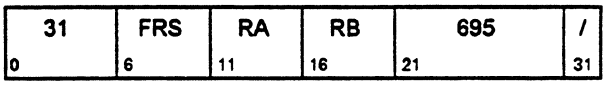
EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Floating-Point Single with Update Indexed X-form

stfsux FRS,RA,RB




```
EA ← (RA) + (RB)
MEM(EA, 4) ← SINGLE(FRS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB).

The contents of register FRS is converted to single format (see page 103) and stored into the word in storage addressed by EA.

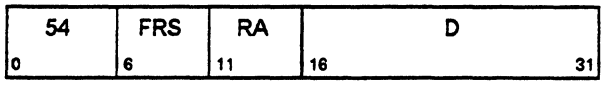
EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Floating-Point Double D-form

stfd FRS,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 8) ← (FRS)
```

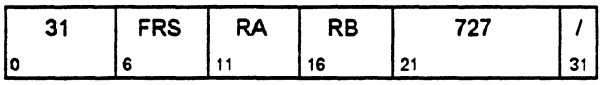
Let the effective address (EA) be the sum (RA|0)+D.

The contents of register FRS is stored into the doubleword in storage addressed by EA.

Special Registers Altered:
None

Store Floating-Point Double Indexed X-form

stfdx FRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (FRS)
```

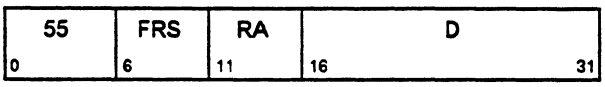
Let the effective address (EA) be the sum (RA|0)+(RB).

The contents of register FRS is stored into the doubleword in storage addressed by EA.

Special Registers Altered:
None

Store Floating-Point Double with Update D-form

stfdu FRS,D(RA)




```
EA ← (RA) + EXTS(D)
MEM(EA, 8) ← (FRS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D.

The contents of register FRS is stored into the doubleword in storage addressed by EA.

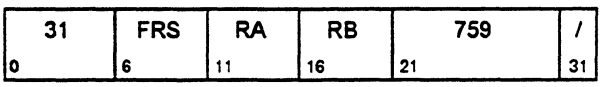
EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Floating-Point Double with Update Indexed X-form

stfdux FRS,RA,RB




```
EA ← (RA) + (RB)
MEM(EA, 8) ← (FRS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB).

The contents of register FRS is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

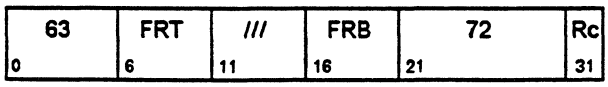
Special Registers Altered:
None

4.6.4 Floating-Point Move Instructions

These instructions copy data from one floating-point register to another with data modifications as described for each instruction. These instructions do not modify the FPSCR.

Floating Move Register X-form

fmr FRT,FRB (Rc=0)
fmr. FRT,FRB (Rc=1)

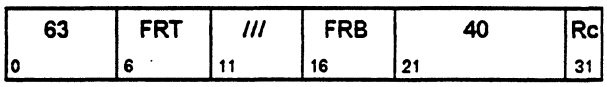


The contents of register FRB is placed into register FRT.

Special Registers Altered:
CR1 (if Rc=1)

Floating Negate X-form

fneg FRT,FRB (Rc=0)
fneg. FRT,FRB (Rc=1)

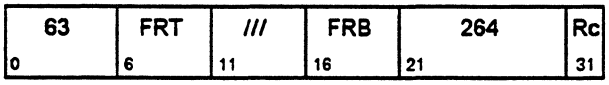


The contents of register FRB with bit 0 inverted is placed into register FRT.

Special Registers Altered:
CR1 (if Rc=1)

Floating Absolute Value X-form

fabs FRT,FRB (Rc=0)
fabs. FRT,FRB (Rc=1)

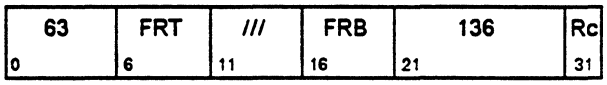


The contents of register FRB with bit 0 set to zero is placed into register FRT.

Special Registers Altered:
CR1 (if Rc=1)

Floating Negative Absolute Value X-form

fnabs FRT,FRB (Rc=0)
fnabs. FRT,FRB (Rc=1)



The contents of register FRB with bit 0 set to one is placed into register FRT.

Special Registers Altered:
CR1 (if Rc=1)

4.6.5 Floating-Point Arithmetic Instructions

Floating Add [Single] A-form

fadd FRT,FRA,FRB (Rc=0)
fadd. FRT,FRA,FRB (Rc=1)

[Power mnemonics: fa, fa.]

63	FRT	FRA	FRB	///	21	Rc
0	6	11	16	21	26	31

fadds FRT,FRA,FRB (Rc=0)
fadds. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	///	21	Rc
0	6	11	16	21	26	31

The floating-point operand in register FRA is added to the floating-point operand in register FRB. 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 FRT.

Floating-point addition 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 an intermediate sum. All 53 bits in the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}=1$.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1 (if Rc=1)

Floating Subtract [Single] A-form

fsub FRT,FRA,FRB (Rc=0)
fsub. FRT,FRA,FRB (Rc=1)

[Power mnemonics: fs, fs.]

63	FRT	FRA	FRB	///	20	Rc
0	6	11	16	21	26	31

fsubs FRT,FRA,FRB (Rc=0)
fsubs. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	///	20	Rc
0	6	11	16	21	26	31

The floating-point operand in register FRB is subtracted from the floating-point operand in register FRA. 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 FRT.

The execution of the *Floating Subtract* instruction is identical to that of *Floating Add*, except that the contents of FRB participates in the operation with its sign bit (bit 0) inverted.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}=1$.

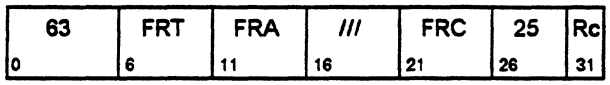
Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1 (if Rc=1)

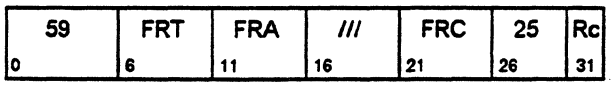
Floating Multiply [Single] A-form

fmul FRT,FRA,FRC (Rc=0)
fmul. FRT,FRA,FRC (Rc=1)

[Power mnemonics: fm, fm.]



fmuls FRT,FRA,FRC (Rc=0)
fmuls. FRT,FRA,FRC (Rc=1)



The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC.

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 FRT.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}=1$.

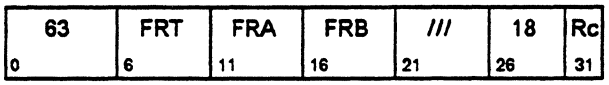
Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXIMZ
CR1 (if Rc=1)

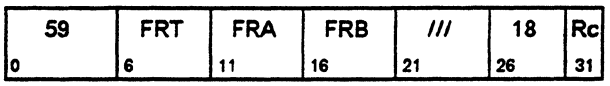
Floating Divide [Single] A-form

fdiv FRT,FRA,FRB (Rc=0)
fdiv. FRT,FRA,FRB (Rc=1)

[Power mnemonics: fd, fd.]



fdivs FRT,FRA,FRB (Rc=0)
fdivs. FRT,FRA,FRB (Rc=1)



The floating-point operand in register FRA is divided by the floating-point operand in register FRB. The remainder is not supplied as a result.

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 FRT.

Floating-point division is based on exponent subtraction and division of the significands.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}=1$ and Zero Divide Exceptions when $FPSCR_{ZE}=1$.

Special Registers Altered:

FPRF FR FI
FX OX UX ZX XX
VXSNAN VXIDI VXZDZ
CR1 (if Rc=1)

4.6.6 Floating-Point Multiply-Add Instructions

These instructions combine a multiply and add operation without an intermediate rounding operation. The fraction part of the intermediate product is 106 bits wide, and all 106 bits take part in the add/subtract portion of the instruction.

Floating Multiply-Add [Single] A-form

fmadd FRT,FRA,FRC,FRB (Rc=0)
fmadd. FRT,FRA,FRC,FRB (Rc=1)

[Power mnemonics: fma, fma.]

63	FRT	FRA	FRB	FRC	29	Rc
0	6	11	16	21	26	31

fmadds FRT,FRA,FRC,FRB (Rc=0)
fmadds. FRT,FRA,FRC,FRB (Rc=1)

59	FRT	FRA	FRB	FRC	29	Rc
0	6	11	16	21	26	31

The operation

FRT ← [(FRA)×(FRC)] + (FRB)

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

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 FRT.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}=1$.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

Floating Multiply-Subtract [Single] A-form

fmsub FRT,FRA,FRC,FRB (Rc=0)
fmsub. FRT,FRA,FRC,FRB (Rc=1)

[Power mnemonics: fms, fms.]

63	FRT	FRA	FRB	FRC	28	Rc
0	6	11	16	21	26	31

fmsubs FRT,FRA,FRC,FRB (Rc=0)
fmsubs. FRT,FRA,FRC,FRB (Rc=1)

59	FRT	FRA	FRB	FRC	28	Rc
0	6	11	16	21	26	31

The operation

FRT ← [(FRA)×(FRC)] − (FRB)

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

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 FRT.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}=1$.

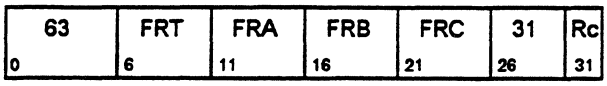
Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

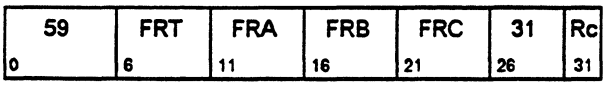
Floating Negative Multiply-Add [Single] A-form

fnmadd FRT,FRA,FRC,FRB (Rc=0)
fnmadd. FRT,FRA,FRC,FRB (Rc=1)

[Power mnemonics: fnma, fnma.]



fnmadds FRT,FRA,FRC,FRB (Rc=0)
fnmadds. FRT,FRA,FRC,FRB (Rc=1)



The operation

FRT ← − ([(FRA)×(FRC)] + (FRB))

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

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, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the *Floating Multiply-Add* instruction and then negating the result, with the following exceptions:

- 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.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the "sign" bit of the SNaN.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}=1$.

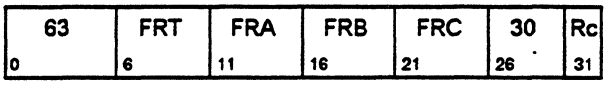
Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

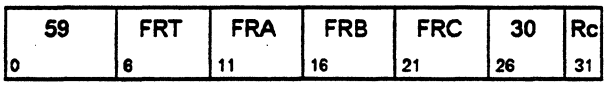
Floating Negative Multiply-Subtract [Single] A-form

fnmsub FRT,FRA,FRC,FRB (Rc=0)
fnmsub. FRT,FRA,FRC,FRB (Rc=1)

[Power mnemonics: fnms, fnms.]



fnmsubs FRT,FRA,FRC,FRB (Rc=0)
fnmsubs. FRT,FRA,FRC,FRB (Rc=1)



The operation

FRT ← − ([(FRA)×(FRC)] − (FRB))

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

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, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the *Floating Multiply-Subtract* instruction and then negating the result, with the following exceptions:

- 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.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the "sign" bit of the SNaN.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}=1$.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

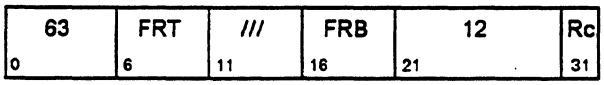
4.6.7 Floating-Point Rounding and Conversion Instructions

Programming Note

Examples of uses of these instructions to perform various conversions can be found in Appendix E.3, "Floating-Point Conversions" on page 159.

Floating Round to Single-Precision X-form

frsp FRT,FRB (Rc=0)
frsp. FRT,FRB (Rc=1)



If it is already in single-precision range, the floating-point operand in register FRB is placed into register FRT. Otherwise the floating-point operand in register FRB is rounded to single-precision using the rounding mode specified by $FPSCR_{RN}$ and placed into register FRT.

The rounding is described fully in Appendix B.1, "Floating-Point Round to Single-Precision Model" on page 123.

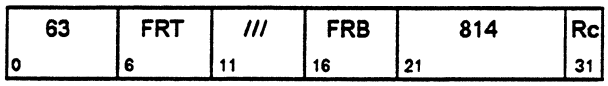
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}=1$.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN
CR1 (if Rc=1)

Floating Convert To Integer Doubleword X-form

fctid FRT,FRB (Rc=0)
fctid. FRT,FRB (Rc=1)



The floating-point operand in register FRB is converted to a 64-bit signed fixed-point integer, using the rounding mode specified by $FPSCR_{RN}$, and placed into register FRT.

If the operand in FRB is greater than $2^{63}-1$, then FRT is set to 0x7FFF_FFFF_FFFF_FFFF. If the operand in FRB is less than -2^{63}, then FRT is set to 0x8000_0000_0000_0000.

The conversion is described fully in Appendix B.2, "Floating-Point Convert to Integer Model" on page 128.

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.

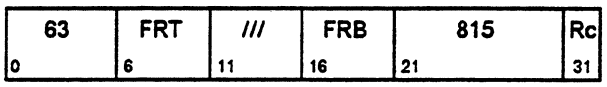
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert To Integer Doubleword with round toward Zero X-form

fctidz FRT,FRB (Rc=0)
fctidz. FRT,FRB (Rc=1)



The floating-point operand in register FRB is converted to a 64-bit signed fixed-point integer, using the rounding mode *Round toward Zero*, and placed into register FRT.

If the operand in FRB is greater than $2^{63}-1$, then FRT is set to 0x7FFF_FFFF_FFFF_FFFF. If the operand in FRB is less than -2^{63}, then FRT is set to 0x8000_0000_0000_0000.

The conversion is described fully in Appendix B.2, "Floating-Point Convert to Integer Model" on page 128.

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.

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.

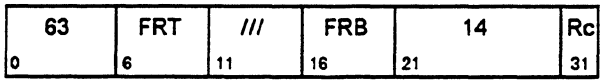
Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert To Integer Word X-form

fctiw FRT,FRB (Rc=0)
fctiw. FRT,FRB (Rc=1)

[Rios-2 mnemonics: fcir, fcir.]



The floating-point operand in register FRB is converted to a 32-bit signed fixed-point integer, using the rounding mode specified by $FPSCR_{RN}$, and placed in bits 32:63 of register FRT. Bits 0:31 of register FRT are undefined.

If the operand in FRB is greater than $2^{31} - 1$, then bits 32:63 of FRT are set to 0x7FFF_FFFF. If the operand in FRB is less than -2^{31}, then bits 32:63 of FRT are set to 0x8000_0000.

The conversion is described fully in Appendix B.2, "Floating-Point Convert to Integer Model" on page 128.

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.

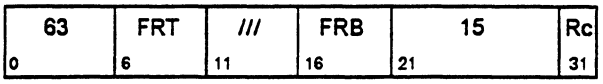
Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert To Integer Word with round toward Zero X-form

fctiwz FRT,FRB (Rc=0)
fctiwz. FRT,FRB (Rc=1)

[Rios-2 mnemonics: fcirz, fcirz.]



The floating-point operand in register FRB is converted to a 32-bit signed fixed-point integer, using the rounding mode *Round toward Zero*, and placed in bits 32:63 of register FRT. Bits 0:31 of register FRT are undefined.

If the operand in FRB is greater than $2^{31} - 1$, then bits 32:63 of FRT are set to 0x7FFF_FFFF. If the operand in FRB is less than -2^{31}, then bits 32:63 of FRT are set to 0x8000_0000.

The conversion is described fully in Appendix B.2, "Floating-Point Convert to Integer Model" on page 128.

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.

Special Registers Altered:

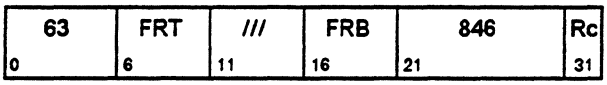
FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Editors' Note

Rios-2 is an unannounced IBM product. If this Book is published before the Rios-2 product is announced, the Rios-2 mnemonics shown for these two instructions (***fctiw*** and ***fctiwz***) should be omitted.

Floating Convert From Integer Doubleword X-form

fcfid FRT,FRB (Rc=0)
fcfid. FRT,FRB (Rc=1)



The 64-bit signed fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. If the result of the conversion is already in double-precision range it is placed into register FRT. Otherwise the result of the conversion is rounded to double-precision using the rounding mode specified by $FPSCR_{RN}$ and placed into register FRT.

The conversion is described fully in Appendix B.3, "Floating-Point Convert from Integer Model" on page 131.

$FPSCR_{FPRF}$ 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.

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.

Special Registers Altered:

FPRF FR FI
FX XX
CR1 (if Rc=1)

4.6.8 Floating-Point Compare Instructions

The floating-point *Compare* instructions compare the contents of two floating-point registers. Comparison ignores the sign of zero (i.e., regards +0 as equal to −0). The comparison can be ordered or unordered.

The comparison sets one bit in the designated CR field to one, and the other three to zero. The FPCC is set in the same way.

The CR field and the FPCC are interpreted as follows:

Bit	Name	Description
0	FL	(FRA) < (FRB)
1	FG	(FRA) > (FRB)
2	FE	(FRA) = (FRB)
3	FU	(FRA) ? (FRB) (unordered)

Floating Compare Unordered X-form

fcmpu BF,FRA,FRB

63	BF	//	FRA	FRB	0	/
0	6	9	11	16	21	31

The floating-point operand in register FRA is compared to the floating-point operand in register FRB. The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signalling, then CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signalling NaN, then VXSNAN is set.

Special Registers Altered:

CR field BF
FPCC
FX
VXSNAN

Floating Compare Ordered X-form

fcmpo BF,FRA,FRB

63	BF	//	FRA	FRB	32	/
0	6	9	11	16	21	31

The floating-point operand in register FRA is compared to the floating-point operand in register FRB. The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signalling, then CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signalling NaN, then VXSNAN is set, and if Invalid Operation is disabled (VE=0) then VXVC is set. Otherwise, if either of the operands is a Quiet NaN then VXVC is set.

Special Registers Altered:

CR field BF
FPCC
FX
VXSNAN VXVC

4.6.9 Floating-Point Status and Control Register Instructions

Every *Floating-Point Status and Control Register* instruction appears to synchronize the effects of all floating-point instructions executed by a given processor. Executing a *Floating-Point Status and Control Register* instruction ensures that all floating-point instructions previously initiated by the given processor appear to have completed before the *Floating-Point Status and Control Register* instruction is initiated, and that no subsequent floating-point instructions appear to be initiated by the given processor until the *Floating-Point Status and Control Register* instruction has completed. In particular:

- all exceptions that will be caused by the previously initiated instructions are recorded in the FPSCR before the *Floating-Point Status and Control Register* instruction is initiated;
- all invocations of the floating-point exception handler that will be caused by the previously initiated instructions have occurred before the *Floating-Point Status and Control Register* instruction is initiated; and
- no subsequent floating-point instruction that depends on or alters the settings of any FPSCR bits appears to be initiated until the *Floating-Point Status and Control Register* instruction has completed.

(Floating-point *Storage Access* instructions are not affected.)

Move From FPSCR X-form

mffs FRT (Rc=0)
mffs. FRT (Rc=1)

63	FRT	///	///	583	Rc
0	6	11	16	21	31

The contents of the FPSCR is placed into bits 32:63 of register FRT. Bits 0:31 of register FRT are undefined.

Special Registers Altered:

CR1 (if Rc=1)

Move to Condition Register from FPSCR X-form

mcrfs BF,BFA

63	BF	//	BFA	//	///	64	/
0	6	9	11	14	16	21	31

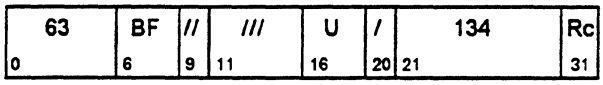
The contents of FPSCR field BFA are copied to CR field BF. All exception bits copied are reset to zero in the FPSCR.

Special Registers Altered:

CR field BF
FX OX (if BFA=0)
UX ZX XX VXSNAN (if BFA=1)
VXISI VXIDI VXZDZ VXIMZ (if BFA=2)
VXVC (if BFA=3)
VXSOFT VXSQRT VXCVI (if BFA=5)

Move To FPSCR Field Immediate X-form

mtfsfi BF,U (Rc=0)
mtfsfi. BF,U (Rc=1)



The value of the U field is placed into FPSCR field BF.

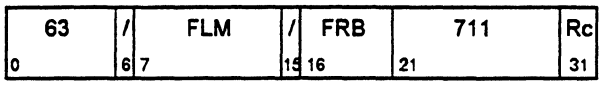
Special Registers Altered:
FPSCR field BF
CR1 (if Rc=1)

Programming Note

When $FPSCR_{0:3}$ is specified, bits 0 (FX) and 3 (OX) are set to the values of U_0 and U_3 (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from U_0 and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 1 and 2 (FEX and VX) are set according to the usual rule, given on page 85, and not from $U_{1:2}$.

Move To FPSCR Fields XFL-form

mtfsf FLM,FRB (Rc=0)
mtfsf. FLM,FRB (Rc=1)



The contents of bits 32:63 of register FRB are placed into the FPSCR under control of the field mask specified by FLM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0-7. If $FLM_i=1$ then FPSCR field i (FPSCR bits 4xi through 4xi+3) is set to the contents of the corresponding field of the low-order 32 bits of register FRB.

Special Registers Altered:
FPSCR fields selected by mask
CR1 (if Rc=1)

Programming Note

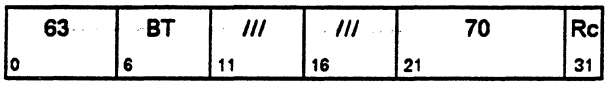
Updating fewer than all eight fields of the FPSCR may have substantially poorer performance on some implementations than updating all the fields.

Programming Note

When $FPSCR_{0:3}$ is specified, bits 0 (FX) and 3 (OX) are set to the values of $(FRB)_{32}$ and $(FRB)_{35}$ (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from $(FRB)_{32}$ and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 1 and 2 (FEX and VX) are set according to the usual rule, given on page 85, and not from $(FRB)_{33:34}$.

Move To FPSCR Bit 0 X-form

mtfsb0 BT (Rc=0)
mtfsb0. BT (Rc=1)



Bit BT of the FPSCR is set to zero.

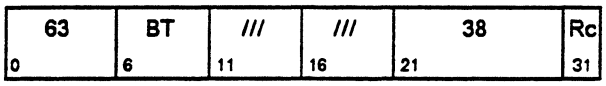
Special Registers Altered:
FPSCR bit BT
CR1 (if Rc=1)

Programming Note

Bits 1 and 2 (FEX and VX) cannot be explicitly reset.

Move To FPSCR Bit 1 X-form

mtfsb1 BT (Rc=0)
mtfsb1. BT (Rc=1)



Bit BT of the FPSCR is set to one.

Special Registers Altered:
FPSCR bit BT
CR1 (if Rc=1)

Programming Note

Bits 1 and 2 (FEX and VX) cannot be explicitly set.

Appendix A. Optional Instructions

The instructions described in this appendix are optional. If an instruction is implemented that matches the semantics of an instruction described here, the implementation should be as specified here.

An implementation may provide all, none, or certain defined groups of these instructions. At present, two such groups are defined:

General Purpose group: ***fsqrt*** and ***fsqrts***.

Graphics group: ***stfiwx, fsel, fres,*** and ***frsqrte***.

A.1 Floating-Point Processor Instructions

A.1.1 Floating-Point Store Instruction

Byte ordering on PowerPC is Big-Endian by default. See Appendix D, "Little-Endian Byte Ordering" on page 145 for the effects of operating a PowerPC system with Little-Endian byte ordering.

Store Floating-Point as Integer Word Indexed X-form

stfiwx FRS,RA,RB

31	FRS	RA	RB	983	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (FRS)32:63
```

Let the effective address (EA) be the sum (RA|0)+(RB).

The contents of the low-order 32 bits of register FRS are stored, without conversion, into the word in storage addressed by EA.

Special Registers Altered:

None

Architecture Note

This instruction is intended for general use and may eventually become part of Chapter 4, Floating-Point Processor.

A.1.2 Floating-Point Arithmetic Instructions

Floating Square Root [Single] A-form

fsqrt FRT,FRB (Rc=0)
fsqrt. FRT,FRB (Rc=1)

63	FRT	///	FRB	///	22	Rc
0	6	11	16	21	26	31

fsqrts FRT,FRB (Rc=0)
fsqrts. FRT,FRB (Rc=1)

59	FRT	///	FRB	///	22	Rc
0	6	11	16	21	26	31

The square root of the floating-point operand in register FRB is placed into register 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 FRT.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
-∞	QNaN[1]	VXSQRT
< 0	QNaN[1]	VXSQRT
-0	-0	None
+∞	+∞	None
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

[1]No result if $FPSCR_{VE}$ = 1.

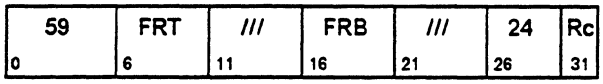
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX XX
VXSNAN VXSQRT
CR1 (if Rc=1)

Floating Reciprocal Estimate Single A-form

fres FRT,FRB (Rc=0)
fres. FRT,FRB (Rc=1)



A single-precision estimate of the reciprocal of the floating-point operand in register FRB is placed into register FRT. The estimate placed into register FRT is correct to a precision of one part in 256 of the reciprocal of (FRB).

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
$-\infty$	-0	None
-0	$-\infty$[1]	ZX
$+0$	$+\infty$[1]	ZX
$+\infty$	$+0$	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

[1]No result if $FPSCR_{ZE} = 1$.
[2]No result if $FPSCR_{VE} = 1$.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}=1$ and Zero Divide Exceptions when $FPSCR_{ZE}=1$.

Special Registers Altered:

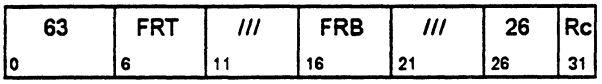
FPRF FR (undefined) FI (undefined)
FX OX UX ZX
VXSNAN
CR1 (if Rc=1)

Architecture Note

No double-precision version of this instruction is provided because graphics applications are expected to need only the single-precision version, and no other important performance-critical applications are expected to need a double-precision version.

Floating Reciprocal Square Root Estimate A-form

frsqrte FRT,FRB (Rc=0)
frsqrte. FRT,FRB (Rc=1)



A double-precision estimate of the reciprocal of the square root of the floating-point operand in register FRB is placed into register 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).

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
$-\infty$	QNaN[2]	VXSQRT
< 0	QNaN[2]	VXSQRT
-0	$-\infty$[1]	ZX
$+0$	$+\infty$[1]	ZX
$+\infty$	$+0$	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

[1]No result if $FPSCR_{ZE} = 1$.
[2]No result if $FPSCR_{VE} = 1$.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}=1$ and Zero Divide Exceptions when $FPSCR_{ZE}=1$.

Special Registers Altered:

FPRF FR (undefined) FI (undefined)
FX ZX
VXSNAN VXSQRT
CR1 (if Rc=1)

Architecture Note

No single-precision version of this instruction is provided because it would be superflous: if (FRB) is representable in single-precision format, then so is (FRT).

A.1.3 Floating-Point Select Instruction

Floating Select A-form

fsel FRT,FRA,FRC,FRB (Rc=0)
fsel. FRT,FRA,FRC,FRB (Rc=1)

63	FRT	FRA	FRB	FRC	23	Rc
0	6	11	16	21	26	31

```
if (FRA) ≥ 0.0 then FRT ← (FRC)
else FRT ← (FRB)
```

The floating-point operand in register FRA is compared to the value zero. If the operand is greater than or equal to zero, register FRT is set to the contents of register FRC. If the operand is less than zero or is a NaN, register FRT is set to the contents of register FRB. The comparison ignores the sign of zero (i.e., regards +0 as equal to –0).

Special Registers Altered:
CR1 (if Rc=1)

Architecture Note

The *Select* instruction is similar to a *Move* instruction, and therefore does not alter FPRF.

Programming Note

Examples of uses of this instruction can be found in Appendices E.3, "Floating-Point Conversions" on page 159, and E.4, "Floating-Point Selection" on page 162.

Warning: Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section E.4.4, "Notes" on page 162.

Appendix B. Suggested Floating-Point Models

B.1 Floating-Point Round to Single-Precision Model

The following describes algorithmically the operation of the *Floating Round to Single-Precision* instruction.

```
If (FRB)1:11 < 897 and (FRB)1:63 > 0 then
    Do
        If FPSCRUE = 0 then goto Disabled Exponent Underflow
        If FPSCRUE = 1 then goto Enabled Exponent Underflow
    End

If (FRB)1:11 > 1150 and (FRB)1:11 < 2047 then
    Do
        If FPSCROE = 0 then goto Disabled Exponent Overflow
        If FPSCROE = 1 then goto Enabled Exponent Overflow
    End

If (FRB)1:11 > 896 and (FRB)1:11 < 1151 then goto Normal Operand

If (FRB)1:63 = 0 then goto Zero Operand

If (FRB)1:11 = 2047 then
    Do
        If (FRB)12:63 = 0 then goto Infinity Operand
        If (FRB)12 = 1 then goto QNaN Operand
        If (FRB)12 = 0 and (FRB)13:63 > 0 then goto SNaN Operand
    End
```

Disabled Exponent Underflow:

```
sign ← (FRB)0
If (FRB)1:11 = 0 then
    Do
        exp ← −1022
        frac ← 0b0 || (FRB)12:63
    End
If (FRB)1:11 > 0 then
    Do
        exp ← (FRB)1:11 − 1023
        frac ← 0b1 || (FRB)12:63
    End
Denormalize operand:
    G || R || X ← 0b000
    Do while exp < −126
        exp ← exp + 1
        frac || G || R || X ← 0b0 || frac || G || (R | X)
    End
FPSCRUX ← frac24:52 || G || R || X > 0
If frac24:52 || G || R || X > 0 then FPSCRXX ← 1
Round single(sign,exp,frac,G,R,X)
If frac = 0 then
    Do
        FRT00 ← sign
        FRT01:63 ← 0
        If sign = 0 then FPSCRFPRF ← "+zero"
        If sign = 1 then FPSCRFPRF ← "−zero"
    End
If frac > 0 then
    Do
        If frac0 = 1 then
            Do
                If sign = 0 then FPSCRFPRF ← "+normal number"
                If sign = 1 then FPSCRFPRF ← "−normal number"
            End
        If frac0 = 0 then
            Do
                If sign = 0 then FPSCRFPRF ← "+denormalized number"
                If sign = 1 then FPSCRFPRF ← "−denormalized number"
            End
        Normalize operand:
            Do while frac0 = 0
                exp ← exp−1
                frac || G || R ← frac1:52 || G || R || 0b0
            End
        FRT0 ← sign
        FRT1:11 ← exp + 1023
        FRT12:63 ← frac1:23 || 29 0
    End
Done
```

***Enabled Exponent Underflow*:**

```
FPSCR_UX ← 1
sign ← (FRB)_0
If (FRB)_1:11 = 0 then
    Do
        exp ← −1022
        frac ← 0b0 || (FRB)_12:63
    End
If (FRB)_1:11 > 0 then
    Do
        exp ← (FRB)_1:11 − 1023
        frac ← 0b1 || (FRB)_12:63
    End
Normalize operand:
    Do while frac_0 = 0
        exp ← exp − 1
        frac ← frac_1:52 || 0b0
    End
If frac_24:52 > 0 then FPSCR_XX ← 1
Round single(sign,exp,frac,0,0,0)
exp ← exp + 192
FRT_0 ← sign
FRT_1:11 ← exp + 1023
FRT_12:63 ← frac_1:23 || 29 0
If sign = 0 then FPSCR_FPRF ← "+normal number"
If sign = 1 then FPSCR_FPRF ← "−normal number"
Done
```

***Disabled Exponent Overflow*:**

```
inc ← 0
FPSCR_OX ← 1
FPSCR_XX ← 1
If FPSCR_RN = 0b00 then              /* Round to Nearest */
    Do
        inc ← 1
        If (FRB)_0 = 0 then FRT ← 0x7FF0_0000_0000_0000
        If (FRB)_0 = 1 then FRT ← 0xFFF0_0000_0000_0000
        If (FRB)_0 = 0 then FPSCR_FPRF ← "+infinity"
        If (FRB)_0 = 1 then FPSCR_FPRF ← "−infinity"
    End
If FPSCR_RN = 0b01 then              /* Round Truncate */
    Do
        If (0b0 || (FRB)_1:63) < 0x47EF_FFFF_E000_0000 then inc ← 1
        If (FRB)_0 = 0 then FRT ← 0x47EF_FFFF_E000_0000
        If (FRB)_0 = 1 then FRT ← 0xC7EF_FFFF_E000_0000
        If (FRB)_0 = 0 then FPSCR_FPRF ← "+normal number"
        If (FRB)_0 = 1 then FPSCR_FPRF ← "−normal number"
    End
If FPSCR_RN = 0b10 then              /* Round to +Infinity */
    Do
        If (FRB)_0 = 0 then inc ← 1
        If ((FRB)_0 = 1) & ((FRB) > 0xC7EF_FFFF_E000_0000) then inc ← 1
        If (FRB)_0 = 0 then FRT ← 0x7FF0_0000_0000_0000
        If (FRB)_0 = 1 then FRT ← 0xC7EF_FFFF_E000_0000
        If (FRB)_0 = 0 then FPSCR_FPRF ← "+infinity"
        If (FRB)_0 = 1 then FPSCR_FPRF ← "−normal number"
    End
If FPSCR_RN = 0b11 then              /* Round to −Infinity */
    Do
        If ((FRB)_0 = 0) & ((FRB) < 0x47EF_FFFF_E000_0000) then inc ← 1
        If (FRB)_0 = 1 then inc ← 1
        If (FRB)_0 = 0 then FRT ← 0x47EF_FFFF_E000_0000
        If (FRB)_0 = 1 then FRT ← 0xFFF0_0000_0000_0000
        If (FRB)_0 = 0 then FPSCR_FPRF ← "+normal number"
        If (FRB)_0 = 1 then FPSCR_FPRF ← "−infinity"
    End
FPSCR_FR ← inc
FPSCR_FI ← 1
Done
```

Enabled Exponent Overflow:

```
    sign ← (FRB)0
    exp ← (FRB)1:11 − 1023
    frac ← 0b1 || (FRB)12:63
    If frac24:52 > 0 then FPSCRXX ← 1
    Round single(sign,exp,frac,0,0,0)
Enabled Overflow:
    FPSCROX ← 1
    exp ← exp − 192
    FRT0 ← sign
    FRT1:11 ← exp + 1023
    FRT12:63 ← frac1:23 || 29 0
    If sign = 0 then FPSCRFPRF ← "+normal number"
    If sign = 1 then FPSCRFPRF ← "−normal number"
    Done
```

Zero Operand:

```
    FRT ← (FRB)
    If (FRB)0 = 0 then FPSCRFPRF ← "+zero"
    If (FRB)0 = 1 then FPSCRFPRF ← "−zero"
    FPSCRFR FI ← 0b00
    Done
```

Infinity Operand:

```
    FRT ← (FRB)
    If (FRB)0 = 0 then FPSCRFPRF ← "+infinity"
    If (FRB)0 = 1 then FPSCRFPRF ← "−infinity"
    FPSCRFR FI ← 0b00
    Done
```

QNaN Operand:

```
    FRT ← (FRB)0:34 || 29 0
    FPSCRFPRF ← "QNaN"
    FPSCRFR FI ← 0b00
    Done
```

SNaN Operand:

```
    FPSCRVXSNAN ← 1
    If FPSCRVE = 0 then
        Do
            FRT0:11 ← (FRB)0:11
            FRT12 ← 1
            FRT13:63 ← (FRB)13:34 || 29 0
            FPSCRFPRF ← "QNaN"
        End
    FPSCRFR FI ← 0b00
    Done
```

Normal Operand:

```
sign ← (FRB)0
exp ← (FRB)1:11 − 1023
frac ← 0b1 || (FRB)12:63
If frac24:52 > 0 then FPSCRXX ← 1
Round single(sign,exp,frac,0,0,0)
If exp > +127 and FPSCROE = 0 then go to Disabled Exponent Overflow
If exp > +127 and FPSCROE = 1 then go to Enabled Overflow
FRT0 ← sign
FRT1:11 ← exp + 1023
FRT12:63 ← frac1:23 || 29 0
If sign = 0 then FPSCRFPRF ← '+normal number'
If sign = 1 then FPSCRFPRF ← '−normal number'
Done
```

Round single(sign,exp,frac,G,R,X):

```
inc ← 0
lsb ← frac23
gbit ← frac24
rbit ← frac25
xbit ← (frac26:52||G||R||X)≠0
If FPSCRRN = 0b00 then
    Do
        If sign || lsb || gbit || rbit || xbit = 0bu11uu then inc ← 1   /* comparison ignores u bits */
        If sign || lsb || gbit || rbit || xbit = 0bu011u then inc ← 1   /* comparison ignores u bits */
        If sign || lsb || gbit || rbit || xbit = 0bu01u1 then inc ← 1   /* comparison ignores u bits */
    End
If FPSCRRN = 0b10 then
    Do
        If sign || lsb || gbit || rbit || xbit = 0b0u1uu then inc ← 1   /* comparison ignores u bits */
        If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc ← 1   /* comparison ignores u bits */
        If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc ← 1   /* comparison ignores u bits */
    End
If FPSCRRN = 0b11 then
    Do
        If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc ← 1   /* comparison ignores u bits */
        If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc ← 1   /* comparison ignores u bits */
        If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc ← 1   /* comparison ignores u bits */
    End
frac0:23 ← frac0:23 + inc
If carry_out = 1 then
    Do
        frac0:23 ← 0b1 || frac0:22
        exp ← exp + 1
    End
FPSCRFR ← inc
FPSCRFI ← gbit | rbit | xbit
Return
```

B.2 Floating-Point Convert to Integer Model

The following describes algorithmically the operation of the *Floating Convert to Integer* instructions.

```
If Floating Convert to Integer Word
    Then Do
        Then round_mode ← FPSCR[RN]
        tgt_precision ← "32-bit integer"
    End

If Floating Convert to Integer Word with round toward Zero
    Then Do
        round_mode ← 0b01
        tgt_precision ← "32-bit integer"
    End

If Floating Convert to Integer Doubleword
    Then Do
        round_mode ← FPSCR[RN]
        tgt_precision ← "64-bit integer"
    End

If Floating Convert to Integer Doubleword with round toward Zero
    Then Do
        round_mode ← 0b01
        tgt_precision ← "64-bit integer"
    End

If (FRB)1:11 = 2047 and (FRB)12:63 = 0 then goto Infinity Operand
If (FRB)1:11 = 2047 and (FRB)12 = 0 then goto SNaN Operand
If (FRB)1:11 = 2047 and (FRB)12 = 1 then goto QNaN Operand
If (FRB)1:11 > 1086 then goto Large Operand

sign ← (FRB)0
If (FRB)1:11 > 0 then exp ← (FRB)1:11 − 1023   /* exp − bias */
If (FRB)1:11 = 0 then exp ← −1022
If (FRB)1:11 > 0 then frac0:64 ← 0b01 || (FRB)12:63 || 11 0   /* normal */
If (FRB)1:11 = 0 then frac0:64 ← 0b00 || (FRB)12:63 || 11 0   /* denormal */

gbit || rbit || xbit ← 0b000
Do i = 1,63−exp   /* do the loop 0 times if exp = 63 */
    frac0:64 || gbit || rbit || xbit ← 0b0 || frac0:64 || gbit || (rbit | xbit)
End

If gbit | rbit | xbit then FPSCR[XX] ← 1

Round Integer(frac,gbit,rbit,xbit,round_mode)

If sign = 1 then frac0:64 ← ¬frac0:64 + 1

If tgt_precision = "32-bit integer" and frac0:64 > +2^31-1 then goto Large Operand
If tgt_precision = "64-bit integer" and frac0:64 > +2^63-1 then goto Large Operand
If tgt_precision = "32-bit integer" and frac0:64 < −2^31 then goto Large Operand
If tgt_precision = "64-bit integer" and frac0:64 < −2^63 then goto Large Operand

If tgt_precision = "32-bit integer" then FRT ← 0xuuuu_uuuu || frac33:64   /* u is undefined hex digit */
If tgt_precision = "64-bit integer" then FRT ← frac1:64
FPSCR[FPRF] ← undefined
Done
```

Round Integer(frac,gbit,rbit,xbit,round_mode):

```
inc ← 0
If round_mode = 0b00 then
    Do
        If sign || frac64 || gbit || rbit || xbit = 0bu11ux then inc ← 1   /* comparison ignores u bits */
        If sign || frac64 || gbit || rbit || xbit = 0bu011x then inc ← 1   /* comparison ignores u bits */
        If sign || frac64 || gbit || rbit || xbit = 0bu01u1 then inc ← 1   /* comparison ignores u bits */
    End
If round_mode = 0b10 then
    Do
        If sign || frac64 || gbit || rbit || xbit = 0b0u1ux then inc ← 1   /* comparison ignores u bits */
        If sign || frac64 || gbit || rbit || xbit = 0b0uu1x then inc ← 1   /* comparison ignores u bits */
        If sign || frac64 || gbit || rbit || xbit = 0b0uuu1 then inc ← 1   /* comparison ignores u bits */
    End
If round_mode = 0b11 then
    Do
        If sign || frac64 || gbit || rbit || xbit = 0b1u1ux then inc ← 1   /* comparison ignores u bits */
        If sign || frac64 || gbit || rbit || xbit = 0b1uu1x then inc ← 1   /* comparison ignores u bits */
        If sign || frac64 || gbit || rbit || xbit = 0b1uuu1 then inc ← 1   /* comparison ignores u bits */
    End
frac0:64 ← frac0:64 + inc
FPSCRFR ← inc
FPSCRFI ← gbit | rbit | xbit
Return
```

Infinity Operand:

```
FPSCRFR FI VXCVI ← 0b001
If FPSCRVE = 0 then Do
    If tgt_precision = "32-bit integer" then
        Do
            If sign = 0 then FRT ← 0xuuuu_uuuu_7FFF_FFFF   /* u is undefined hex digit */
            If sign = 1 then FRT ← 0xuuuu_uuuu_8000_0000   /* u is undefined hex digit */
        End
    Else
        Do
            If sign = 0 then FRT ← 0x7FFF_FFFF_FFFF_FFFF
            If sign = 1 then FRT ← 0x8000_0000_0000_0000
        End
    FPSCRFPRF ← undefined
    End
Done
```

SNaN Operand:

```
FPSCRFR FI VXCVI VXSNAN ← 0b0011
If FPSCRVE = 0 then
    Do
        If tgt_precision = "32-bit integer" then FRT ← 0xuuuu_uuuu_8000_0000   /* u is undefined hex digit */
        If tgt_precision = "64-bit integer" then FRT ← 0x8000_0000_0000_0000
        FPSCRFPRF ← undefined
    End
Done
```

QNaN Operand:

```
FPSCR[FR FI VXCVI] ← 0b001
If FPSCR[VE] = 0 then
    Do
        If tgt_precision = "32-bit integer" then FRT ← 0xuuuu_uuuu_8000_0000   /* u is undefined hex digit */
        If tgt_precision = "64-bit integer" then FRT ← 0x8000_0000_0000_0000
        FPSCR[FPRF] ← undefined
    End
Done
```

Large Operand:

```
FPSCR[FR FI VXCVI] ← 0b001
If FPSCR[VE] = 0 then Do
    If tgt_precision = "32-bit integer" then
        Do
            If sign = 0 then FRT ← 0xuuuu_uuuu_7FFF_FFFF   /* u is undefined hex digit */
            If sign = 1 then FRT ← 0xuuuu_uuuu_8000_0000   /* u is undefined hex digit */
        End
    Else
        Do
            If sign = 0 then FRT ← 0x7FFF_FFFF_FFFF_FFFF
            If sign = 1 then FRT ← 0x8000_0000_0000_0000
        End
    FPSCR[FPRF] ← undefined
    End
Done
```

B.3 Floating-Point Convert from Integer Model

The following describes algorithmically the operation of the *Floating Convert from Integer* instructions.

```
sign ← (FRB)0
exp ← 63
frac0:63 ← (FRB)

If frac0:63 = 0 then go to Zero Operand

If sign = 1 then frac0:63 ← ¬frac0:63 + 1

Do until frac0 = 1
   frac0:63 ← frac1:63 || 0b0
   exp ← exp – 1
End

Round Float(sign,exp,frac,FPSCRRN)

If sign = 1 then FPSCRFPRF ← "–normal number"
If sign = 0 then FPSCRFPRF ← "+normal number"
FRT0 ← sign
FRT1:11 ← exp + 1023   /* exp + bias */
FRT12:63 ← frac1:52
Done
```

Zero Operand:

```
FPSCRFR FI ← 0b00
FPSCRFPRF ← "+zero"
FRT ← 0x0000_0000_0000_0000
Done
```

***Round Float**(sign,exp,frac,round_mode)*:

```
inc ← 0
lsb ← frac52
gbit ← frac53
rbit ← frac54
xbit ← frac55:63 > 0
If round_mode = 0b00 then
    Do
        If sign || lsb || gbit || rbit || xbit = 0bu11uu then inc ← 1   /* comparison ignores u bits */
        If sign || lsb || gbit || rbit || xbit = 0bu011u then inc ← 1   /* comparison ignores u bits */
        If sign || lsb || gbit || rbit || xbit = 0bu01u1 then inc ← 1   /* comparison ignores u bits */
    End
If round_mode = 0b10 then
    Do
        If sign || lsb || gbit || rbit || xbit = 0b0u1uu then inc ← 1   /* comparison ignores u bits */
        If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc ← 1   /* comparison ignores u bits */
        If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc ← 1   /* comparison ignores u bits */
    End
If round_mode = 0b11 then
    Do
        If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc ← 1   /* comparison ignores u bits */
        If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc ← 1   /* comparison ignores u bits */
        If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc ← 1   /* comparison ignores u bits */
    End
frac0:52 ← frac0:52 + inc
If carry_out = 1 then exp ← exp + 1
FPSCRFR ← inc
FPSCRFI ← gbit | rbit | xbit
If (gbit | rbit | xbit) then FPSCRXX ← 1
Return
```

Appendix C. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided that defines simple shorthand for the most frequently used forms of *Branch Conditional, Compare, Trap, Rotate and Shift,* and certain other instructions.

PowerPC-compliant assemblers will provide the mnemonics and symbols listed here, and possibly others. Programs written to be portable across various assemblers for the PowerPC Architecture should not assume the existence of mnemonics not defined in the PowerPC Architecture Books.

C.1 Branch mnemonics

The mnemonics discussed in this section are variations of the *Branch Conditional* instructions.

C.1.1 BO and BI fields

The 5-bit BO field in *Branch Conditional* instructions encodes the following operations:

- Decrement CTR
- Test CTR equal to 0
- Test CTR not equal to 0
- Test condition true
- Test condition false
- Branch prediction (taken, fall through)

The 5-bit BI field in *Branch Conditional* instructions specifies which of the 32 bits in the CR represents the condition to test.

To provide an extended mnemonic for every possible combination of BO and BI fields would require $2^{10} = 1024$ mnemonics. Most of these would be only marginally useful. The following abbreviated set is intended to cover the most useful cases. Unusual cases can be coded using a basic *Branch Conditional* mnemonic (***bc, bclr, bcctr***) with the condition to be tested specified as a numeric operand.

C.1.2 Simple branch mnemonics

The mnemonics in Table 2 allow all the useful BO encodings to be specified, along with the AA (absolute address) and LK (set Link Register) fields.

Notice that there are no extended mnemonics for relative and absolute unconditional branches. For these the basic mnemonics ***b, ba, bl,*** and ***bla*** should be used.

Table 2. Simple branch mnemonics

Branch semantics	LR not set				LR set			
	bc Relative	*bca* Absolute	*bclr* To LR	*bcctr* To CTR	*bcl* Relative	*bcla* Absolute	*bclrl* To LR	*bcctrl* To CTR
Branch unconditionally	–	–	blr	bctr	–	–	blrl	bctrl
Branch if condition true	bt	bta	btlr	btctr	btl	btla	btlrl	btctrl
Branch if condition false	bf	bfa	bflr	bfctr	bfl	bfla	bflrl	bfctrl
Decrement CTR, branch if CTR non-zero	bdnz	bdnza	bdnzlr	–	bdnzl	bdnzla	bdnzlrl	–
Decrement CTR, branch if CTR non-zero AND condition true	bdnzt	bdnzta	bdnztlr	–	bdnztl	bdnztla	bdnztlrl	–
Decrement CTR, branch if CTR non-zero AND condition false	bdnzf	bdnzfa	bdnzflr	–	bdnzfl	bdnzfla	bdnzflrl	–
Decrement CTR, branch if CTR zero	bdz	bdza	bdzlr	–	bdzl	bdzla	bdzlrl	–
Decrement CTR, branch if CTR zero AND condition true	bdzt	bdzta	bdztlr	–	bdztl	bdztla	bdztlrl	–
Decrement CTR, branch if CTR zero AND condition false	bdzf	bdzfa	bdzflr	–	bdzfl	bdzfla	bdzflrl	–

Instructions using one of the mnemonics in Table 2 that tests a condition specify the condition as the first operand of the instruction. The following symbols are defined for use in such an operand. They can be combined with other values in an expression that identifies the CR bit (0:31) to be tested. These symbols and expressions can also be used with the basic *Branch Conditional* mnemonics, to specify the BI field.

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

Examples

1. Decrement CTR and branch if it is still non-zero (closure of a loop controlled by a count loaded into CTR).

   ```
   bdnz   target                    (equivalent to:   bc   16,0,target)
   ```

2. Same as (1) but branch only if CTR is non zero and condition in CR0 is "equal."

   ```
   bdnzt  eq,target                 (equivalent to:   bc   8,2,target)
   ```

3. Same as (2), but "equal" condition is in CR5.

   ```
   bdnzt  4*cr5+eq,target           (equivalent to:   bc   8,22,target)
   ```

4. Branch if bit 27 of CR is false.

   ```
   bf     27,target                 (equivalent to:   bc   4,27,target)
   ```

5. Same as (4), but set the Link Register. This is a form of conditional "call."

   ```
   bfl    27,target                 (equivalent to:   bcl  4,27,target)
   ```

C.1.3 Branch mnemonics incorporating conditions

The mnemonics defined in Table 3 on page 136 are variations of the "branch if condition true" and "branch if condition false" BO encodings, with the most useful values of BI represented in the mnemonic rather than specified as a numeric operand.

A standard set of codes has been adopted for the most common combinations of branch conditions.

Code	Meaning
lt	Less than
le	Less than or equal
eq	Equal
ge	Greater than or equal
gt	Greater than
nl	Not less than
ne	Not equal
ng	Not greater than
so	Summary overflow
ns	Not summary overflow
un	Unordered (after floating-point comparison)
nu	Not unordered (after floating-point comparison)

These codes are reflected in the mnemonics shown in Table 3 on page 136.

Table 3. Branch mnemonics incorporating conditions								
Branch semantics	**LR not set**				**LR set**			
	bc **Relative**	***bca*** **Absolute**	***bclr*** **To LR**	***bcctr*** **To CTR**	***bcl*** **Relative**	***bcla*** **Absolute**	***bclrl*** **To LR**	***bcctrl*** **To CTR**
Branch if less than	blt	blta	bltlr	bltctr	bltl	bltla	bltlrl	bltctrl
Branch if less than or equal	ble	blea	blelr	blectr	blel	blela	blelrl	blectrl
Branch if equal	beq	beqa	beqlr	beqctr	beql	beqla	beqlrl	beqctrl
Branch if greater than or equal	bge	bgea	bgelr	bgectr	bgel	bgela	bgelrl	bgectrl
Branch if greater than	bgt	bgta	bgtlr	bgtctr	bgtl	bgtla	bgtlrl	bgtctrl
Branch if not less than	bnl	bnla	bnllr	bnlctr	bnll	bnlla	bnllrl	bnlctrl
Branch if not equal	bne	bnea	bnelr	bnectr	bnel	bnela	bnelrl	bnectrl
Branch if not greater than	bng	bnga	bnglr	bngctr	bngl	bngla	bnglrl	bngctrl
Branch if summary overflow	bso	bsoa	bsolr	bsoctr	bsol	bsola	bsolrl	bsoctrl
Branch if not summary overflow	bns	bnsa	bnslr	bnsctr	bnsl	bnsla	bnslrl	bnsctrl
Branch if unordered	bun	buna	bunlr	bunctr	bunl	bunla	bunlrl	bunctrl
Branch if not unordered	bnu	bnua	bnulr	bnuctr	bnul	bnula	bnulrl	bnuctrl

Instructions using the mnemonics in Table 3 specify the Condition Register field in an optional first operand. If the CR field being tested is CR0, this operand need not be specified. Otherwise, one of the CR field symbols listed earlier is coded as the first operand.

Examples

1. Branch if CR0 reflects condition "not equal."

   bne target (equivalent to: bc 4,2,target)

2. Same as (1), but condition is in CR3.

   bne cr3,target (equivalent to: bc 4,14,target)

3. Branch to an absolute target if CR4 specifies "greater than," setting the Link Register. This is a form of conditional "call."

   bgtla cr4,target (equivalent to: bcla 12,17,target)

4. Same as (3), but target address is in the Count Register.

   bgtctrl cr4 (equivalent to: bcctrl 12,17)

C.1.4 Branch prediction

In *Branch Conditional* instructions that are not always taken, the low-order bit ("y" bit) of the BO field provides a hint about whether the branch is likely to be taken: see the discussion of the "y" bit in Section 2.4.1, Branch Instructions, on page 18.

PowerPC-compliant assemblers set this bit to 0 unless otherwise directed. This default action means that:

- A *Branch Conditional* with a negative displacement field is predicted to be taken.
- A *Branch Conditional* with a non-negative displacement field is predicted *not* to be taken (fall through).
- A *Branch Conditional* to an address in the LR or CTR is predicted *not* to be taken (fall through).

If the likely outcome (branch or fall through) of a given *Branch Conditional* instruction is known, a suffix can be added to the mnemonic that tells the assembler how to set the "y" bit.

+ Predict branch to be taken.

− Predict branch *not* to be taken.

Such a suffix can be added to any *Branch Conditional* mnemonic, either basic or extended.

For relative and absolute branches (***bc[l][a]***), the setting of the "y" bit depends on whether the displacement field is negative or non-negative. For negative displacement fields, coding the suffix "+" causes the bit to be set to 0, and coding the suffix "−" causes the bit to be set to 1. For non-negative displacement fields, coding the suffix "+" causes the bit to be set to 1, and coding the suffix "−" causes the bit to be set to 0.

For branches to an address in the LR or CTR (***bclr[l]*** or ***bcctr[l]***), coding the suffix "+" causes the "y" bit to be set to 1, and coding the suffix "−" causes the bit to be set to 0.

Examples

1. Branch if CR0 reflects condition "less than," specifying that the branch should be predicted to be taken.

   blt+ target

2. Same as (1), but target address is in the Link Register and the branch should be predicted not to be taken.

   bltlr−

C.2 Condition Register logical mnemonics

The *Condition Register Logical* instructions can be used to set (to 1), clear (to 0), copy, or invert a given Condition Register bit. Extended mnemonics are provided that allow these operations to be coded easily.

Table 4. Condition Register logical mnemonics

Operation	Extended mnemonic	Equivalent to
Condition Register set	crset bx	creqv bx,bx,bx
Condition Register clear	crclr bx	crxor bx,bx,bx
Condition Register move	crmove bx,by	cror bx,by,by
Condition Register not	crnot bx,by	crnor bx,by,by

Examples

1. Set CR bit 25.

   crset 25 (equivalent to: creqv 25,25,25)

2. Clear the SO bit of CR0.

   crclr so (equivalent to: crxor 3,3,3)

3. Same as (2), but SO bit to be cleared is in CR3.

   crclr 4*cr3+so (equivalent to: crxor 15,15,15)

4. Invert the EQ bit.

   crnot eq,eq (equivalent to: crnor 2,2,2)

5. Same as (4), but EQ bit to be inverted is in CR4, and the result is to be placed into the EQ bit of CR5.

   crnot 4*cr5+eq,4*cr4+eq (equivalent to: crnor 22,18,18)

C.3 Subtract mnemonics

C.3.1 Subtract Immediate

Although there is no "Subtract Immediate" instruction, its effect can be achieved by using an *Add Immediate* instruction with the immediate operand negated. Extended mnemonics are provided that include this negation, making the intent of the computation clearer.

subi	Rx,Ry,value	(equivalent to:	addi Rx,Ry,−value)
subis	Rx,Ry,value	(equivalent to:	addis Rx,Ry,−value)
subic	Rx,Ry,value	(equivalent to:	addic Rx,Ry,−value)
subic.	Rx,Ry,value	(equivalent to:	addic. Rx,Ry,−value)

C.3.2 Subtract

The *Subtract From* instructions subtract the second operand (RA) from the third (RB). Extended mnemonics are provided that use the more "normal" order, in which the third operand is subtracted from the second. Both these mnemonics can be coded with a final "o" and/or "." to cause the OE and/or Rc bit to be set in the underlying instruction.

sub	Rx,Ry,Rz	(equivalent to:	subf Rx,Rz,Ry)
subc	Rx,Ry,Rz	(equivalent to:	subfc Rx,Rz,Ry)

C.4 Compare mnemonics

The L field in the fixed-point *Compare* instructions controls whether the operands are treated as 64-bit quantities (L=1) or as 32-bit quantities (L=0). Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

The BF field can be omitted if the result of the comparison is to be placed in CR Field 0. Otherwise the target CR field must be specified as the first operand, using one of the CR field symbols listed above or an explicit field number.

Note: The basic *Compare* mnemonics of PowerPC are the same as those of Power, but the Power instructions have three operands while the PowerPC instructions have four. The assembler will recognize a basic *Compare* mnemonic with three operands as the Power form, and will generate the instruction with L=0. (Thus the assembler must require that the BF field, which normally can be omitted when CR Field 0 is the target, be specified explicitly if L is.)

C.4.1 Doubleword comparisons

These operations are available only in 64-bit implementations.

Table 5. Doubleword compare mnemonics		
Operation	**Extended mnemonic**	**Equivalent to**
Compare doubleword immediate	cmpdi bf,ra,si	cmpi bf,1,ra,si
Compare doubleword	cmpd bf,ra,rb	cmp bf,1,ra,rb
Compare logical doubleword immediate	cmpldi bf,ra,ui	cmpli bf,1,ra,ui
Compare logical doubleword	cmpld bf,ra,rb	cmpl bf,1,ra,rb

Examples

1. Compare logical (unsigned) 64 bits in register Rx with immediate value 100 and place result in CR0.

   cmpldi Rx,100 (equivalent to: cmpli 0,1,Rx,100)

2. Same as (1), but place results in CR4.

   cmpldi cr4,Rx,100 (equivalent to: cmpli 4,1,Rx,100)

3. Compare registers Rx and Ry as signed 64-bit quantities and place result in CR0.

   cmpd Rx,Ry (equivalent to: cmp 0,1,Rx,Ry)

C.4.2 Word comparisons

These operations are available in all implementations.

Table 6. Word compare mnemonics		
Operation	**Extended mnemonic**	**Equivalent to**
Compare word immediate	cmpwi bf,ra,si	cmpi bf,0,ra,si
Compare word	cmpw bf,ra,rb	cmp bf,0,ra,rb
Compare logical word immediate	cmplwi bf,ra,ui	cmpli bf,0,ra,ui
Compare logical word	cmplw bf,ra,rb	cmpl bf,0,ra,rb

Examples

1. Compare 32 bits in register Rx with immediate value 100 and place result in CR0.

   cmpwi Rx,100 (equivalent to: cmpi 0,0,Rx,100)

2. Same as (1), but place results in CR4.

   cmpwi cr4,Rx,100 (equivalent to: cmpi 4,0,Rx,100)

3. Compare registers Rx and Ry as logical 32-bit quantities and place result in CR0.

   cmplw Rx,Ry (equivalent to: cmpl 0,0,Rx,Ry)

C.5 Trap mnemonics

The mnemonics defined in Table 7 are variations of the *Trap* instructions, with the most useful values of TO represented in the mnemonic rather than specified as a numeric operand.

A standard set of codes has been adopted for the most common combinations of trap conditions.

Code	Meaning	TO encoding	<	>	=	$<^u$	$>^u$
lt	Less than	16	1	0	0	0	0
le	Less than or equal	20	1	0	1	0	0
eq	Equal	4	0	0	1	0	0
ge	Greater than or equal	12	0	1	1	0	0
gt	Greater than	8	0	1	0	0	0
nl	Not less than	12	0	1	1	0	0
ne	Not equal	24	1	1	0	0	0
ng	Not greater than	20	1	0	1	0	0
llt	Logically less than	2	0	0	0	1	0
lle	Logically less than or equal	6	0	0	1	1	0
lge	Logically greater than or equal	5	0	0	1	0	1
lgt	Logically greater than	1	0	0	0	0	1
lnl	Logically not less than	5	0	0	1	0	1
lng	Logically not greater than	6	0	0	1	1	0
(none)	Unconditional	31	1	1	1	1	1

These codes are reflected in the mnemonics shown in Table 7.

Table 7. Trap mnemonics

Trap semantics	64-bit comparison		32-bit comparison	
	tdi Immediate	*td* Register	*twi* Immediate	*tw* Register
Trap unconditionally	–	–	–	trap
Trap if less than	tdlti	tdlt	twlti	twlt
Trap if less than or equal	tdlei	tdle	twlei	twle
Trap if equal	tdeqi	tdeq	tweqi	tweq
Trap if greater than or equal	tdgei	tdge	twgei	twge
Trap if greater than	tdgti	tdgt	twgti	twgt
Trap if not less than	tdnli	tdnl	twnli	twnl
Trap if not equal	tdnei	tdne	twnei	twne
Trap if not greater than	tdngi	tdng	twngi	twng
Trap if logically less than	tdllti	tdllt	twllti	twllt
Trap if logically less than or equal	tdllei	tdlle	twllei	twlle
Trap if logically greater than or equal	tdlgei	tdlge	twlgei	twlge
Trap if logically greater than	tdlgti	tdlgt	twlgti	twlgt
Trap if logically not less than	tdlnli	tdlnl	twlnli	twlnl
Trap if logically not greater than	tdlngi	tdlng	twlngi	twlng

Examples

1. Trap if 64-bit register Rx is not 0.

```
tdnei   Rx,0                    (equivalent to:    tdi    24,Rx,0)
```

2. Same as (1), but comparison is to register Ry.

   tdne Rx,Ry (equivalent to: td 24,Rx,Ry)

3. Trap if register Rx, considered as a 32-bit quantity, is logically greater than 0x7FF.

   twlgti Rx,0x7FF (equivalent to: twi 1,Rx,0x7FF)

4. Trap unconditionally.

   trap (equivalent to: tw 31,0,0)

C.6 Rotate and Shift mnemonics

The *Rotate and Shift* instructions provide powerful and general ways to manipulate register contents, but can be difficult to understand. Extended mnemonics are provided that allow some of the simpler operations to be coded easily.

Mnemonics are provided for the following types of operation:

Extract Select a field of *n* bits starting at bit position *b* in the source register; right or left justify this field in the target register; clear all other bits of the target register to 0.

Insert Select a left-justified or right-justified field of *n* bits in the source register; insert this field starting at bit position *b* of the target register; leave other bits of the target register unchanged. (No extended mnemonic is provided for insertion of a left-justified field when operating on doublewords, because such an insertion requires more than one instruction.)

Rotate Rotate the contents of a register right or left *n* bits without masking.

Shift Shift the contents of a register right or left *n* bits, clearing vacated bits to 0 (logical shift).

Clear Clear the leftmost or rightmost *n* bits of a register to 0.

Clear left and shift left
Clear the leftmost *b* bits of a register, then shift the register left by *n* bits. This operation can be used to scale a (known non-negative) array index by the width of an element.

C.6.1 Operations on doublewords

These operations are available only in 64-bit implementations. All these mnemonics can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

Table 8. Doubleword rotate and shift mnemonics

Operation	Extended mnemonic	Equivalent to
Extract and left justify immediate	extldi ra,rs,*n*,*b*	rldicr ra,rs,*b*,*n*−1
Extract and right justify immediate	extrdi ra,rs,*n*,*b*	rldicl ra,rs,*b*+*n*,64−*n*
Insert from right immediate	insrdi ra,rs,*n*,*b*	rldimi ra,rs,64−(*b*+*n*),*b*
Rotate left immediate	rotldi ra,rs,*n*	rldicl ra,rs,*n*,0
Rotate right immediate	rotrdi ra,rs,*n*	rldicl ra,rs,64−*n*,0
Rotate left	rotld ra,rs,rb	rldcl ra,rs,rb,0
Shift left immediate	sldi ra,rs,*n*	rldicr ra,rs,*n*,63−*n*
Shift right immediate	srdi ra,rs,*n*	rldicl ra,rs,64−*n*,*n*
Clear left immediate	clrldi ra,rs,*n*	rldicl ra,rs,0,*n*
Clear right immediate	clrrdi ra,rs,*n*	rldicr ra,rs,0,63−*n*
Clear left and shift left immediate	clrsldi ra,rs,*b*,*n*	rldic ra,rs,*n*,*b*−*n*

Examples

1. Extract the sign bit (bit 0) of register Ry and place the result right-justified into register Rx.

   extrdi Rx,Ry,1,0 (equivalent to: rldicl Rx,Ry,1,63)

2. Insert the bit extracted in (1) into the sign bit (bit 0) of register Rz.

   insrdi Rz,Rx,1,0 (equivalent to: rldimi Rz,Rx,63,0)

3. Shift the contents of register Rx left 8 bits.

   sldi Rx,Rx,8 (equivalent to: rldicr Rx,Rx,8,55)

4. Clear the high-order 32 bits of Ry and place the result into Rx.

   clrldi Rx,Ry,32 (equivalent to: rldicl Rx,Ry,0,32)

C.6.2 Operations on words

These operations are available in all implementations. All these mnemonics can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

Table 9. Word rotate and shift mnemonics

Operation	Extended mnemonic	Equivalent to
Extract and left justify immediate	extlwi ra,rs,*n*,*b*	rlwinm ra,rs,*b*,0,*n*−1
Extract and right justify immediate	extrwi ra,rs,*n*,*b*	rlwinm ra,rs,*b*+*n*,32−*n*,31
Insert from left immediate	inslwi ra,rs,*n*,*b*	rlwimi ra,rs,32−*b*,*b*,(*b*+*n*)−1
Insert from right immediate	insrwi ra,rs,*n*,*b*	rlwimi ra,rs,32−(*b*+*n*),*b*,(*b*+*n*)−1
Rotate left immediate	rotlwi ra,rs,*n*	rlwinm ra,rs,*n*,0,31
Rotate right immediate	rotrwi ra,rs,*n*	rlwinm ra,rs,32−*n*,0,31
Rotate left	rotlw ra,rs,rb	rlwnm ra,rs,rb,0,31
Shift left immediate	slwi ra,rs,*n*	rlwinm ra,rs,*n*,0,31−*n*
Shift right immediate	srwi ra,rs,*n*	rlwinm ra,rs,32−*n*,*n*,31
Clear left immediate	clrlwi ra,rs,*n*	rlwinm ra,rs,0,*n*,31
Clear right immediate	clrrwi ra,rs,*n*	rlwinm ra,rs,0,0,31−*n*
Clear left and shift left immediate	clrslwi ra,rs,*b*,*n*	rlwinm ra,rs,*n*,*b*−*n*,31−*n*

Examples

1. Extract the sign bit (bit 32) of register Ry and place the result right-justified into register Rx.

   extrwi Rx,Ry,1,0 (equivalent to: rlwinm Rx,Ry,1,31,31)

2. Insert the bit extracted in (1) into the sign bit (bit 32) of register Rz.

   insrwi Rz,Rx,1,0 (equivalent to: rlwimi Rz,Rx,31,0,0)

3. Shift the contents of register Rx left 8 bits, clearing the high-order 32 bits.

   slwi Rx,Rx,8 (equivalent to: rlwinm Rx,Rx,8,0,23)

4. Clear the high-order 16 bits of the low-order 32 bits of Ry and place the result into Rx, clearing the high-order 32 bits of Rx.

   clrlwi Rx,Ry,16 (equivalent to: rlwimn Rx,Ry,0,16,31)

C.7 Move To/From Special Purpose Register mnemonics

The ***mtspr*** and ***mfspr*** instructions specify a Special Purpose Register (SPR) as a numeric operand. Extended mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as a numeric operand.

Table 10. Extended mnemonics for moving to/from an SPR

Special Purpose Register	Move To SPR		Move From SPR	
	Extended	*Equivalent to*	*Extended*	*Equivalent to*
Fixed-Point Exception Register (XER)	mtxer Rx	mtspr 1,Rx	mfxer Rx	mfspr Rx,1
Link Register (LR)	mtlr Rx	mtspr 8,Rx	mflr Rx	mfspr Rx,8
Count Register (CTR)	mtctr Rx	mtspr 9,Rx	mfctr Rx	mfspr Rx,9

Examples

1. Copy the contents of the low-order 32 bits of Rx to the XER.

   mtxer Rx (equivalent to: mtspr 1,Rx)

2. Copy the contents of the LR to register Rx.

   mflr Rx (equivalent to: mfspr Rx,8)

3. Copy the contents of Rx to the CTR.

   mtctr Rx (equivalent to: mtspr 9,Rx)

C.8 Miscellaneous mnemonics

No-op

Many PowerPC instructions can be coded in a way such that, effectively, no operation is performed. An extended mnemonic is provided for the "preferred" form of no-op. If an implementation performs any type of run-time optimization related to no-ops, the preferred form is the no-op that will trigger this.

```
nop          (equivalent to:  ori  0,0,0)
```

Load Immediate

The ***addi*** and ***addis*** instructions can be used to load an immediate value into a register. Extended mnemonics are provided to convey the idea that no addition is being performed but merely data movement (from the immediate field of the instruction to a register).

Load a 16-bit signed immediate value into register Rx:

```
li    Rx,value    (equivalent to:  addi  Rx,0,value)
```

Load a 16-bit signed immediate value, shifted left by 16 bits, into register Rx:

```
lis   Rx,value    (equivalent to:  addis Rx,0,value)
```

Load Address

This mnemonic permits computing the value of a base-displacement operand, using the ***addi*** instruction which normally requires separate register and immediate operands.

```
la      Rx,D(Ry)                    (equivalent to:     addi    Rx,Ry,D)
```

The ***la*** mnemonic is useful for obtaining the address of a variable specified by name, allowing the assembler to supply the base register number and compute the displacement. If the variable v is located at offset Dv bytes from the address in register Rv, and the assembler has been told to use register Rv as a base for references to the data structure containing v, then the following line causes the address of v to be loaded into register Rx.

```
la      Rx,v                        (equivalent to:     addi    Rx,Rv,Dv)
```

Move Register

Several PowerPC instructions can be coded in a way such that they simply copy the contents of one register to another. An extended mnemonic is provided to convey the idea that no computation is being performed but merely data movement (from one register to another).

The following instruction copies the contents of register Ry into register Rx. This mnemonic can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

```
mr      Rx,Ry                       (equivalent to:     or      Rx,Ry,Ry)
```

Complement Register

Several PowerPC instructions can be coded in a way such that they complement the contents of one register and place the result into another register. An extended mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of register Ry and places the result into register Rx. This mnemonic can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

```
not     Rx,Ry                       (equivalent to:     nor     Rx,Ry,Ry)
```

Appendix D. Little-Endian Byte Ordering

> It is computed that eleven Thousand Persons have, at several Times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large Volumes have been published upon this Controversy.
>
> Jonathan Swift, *Gulliver's Travels*

D.1 Byte Ordering

If scalars (individual computational data items) were indivisible, then there would be no such concept as "byte ordering." It is meaningless to talk of the "order" of bits or groups of bits within the smallest addressable unit of storage, because nothing can be observed about such order. Only when scalars, which the programmer and processor regard as indivisible quantities, can be made up of more than one addressable unit of storage does the question of "order" arise.

For a machine in which the smallest addressable unit is the 64-bit doubleword, there is no question of the ordering of "bytes" within doublewords. All scalar transfers between registers and storage are for doublewords, and the address of the "byte" containing the high-order 8 bits of a scalar is no different from the address of a "byte" containing any other part of the scalar.

For PowerPC, as for most computers currently, the smallest addressable storage unit of storage is the 8-bit byte. Most computational scalars are made up of groups of bytes (halfwords, words, doublewords). When a 32-bit scalar is moved from a register to storage, the scalar occupies four consecutive byte addresses. It thus becomes meaningful to discuss the order of the byte addresses with respect to the value of the scalar: which byte contains the highest-order 8 bits of the scalar, which byte contains the next-highest-order 8 bits, and so on.

Given a scalar that spans multiple bytes, the choice of byte ordering is essentially arbitrary. There are $4! = 24$ ways to specify the ordering of four bytes within a word, but only two of these orderings are sensible:

- The ordering that assigns the lowest address to the highest-order ("leftmost") 8 bits of the scalar, the next sequential address to the next-highest-order 8 bits, and so on. This is called **Big-Endian** because the "big end" of the scalar, considered as a binary number, comes first in storage. IBM RISC System/6000, IBM System/370, and Motorola 680x0 are examples of computers using this byte ordering.
- The ordering that assigns the lowest address to the lowest-order ("rightmost") 8 bits of the scalar, the next sequential address to the next-lowest-order 8 bits, and so on. This is called **Little-Endian** because the "little end" of the scalar, considered as a binary number, comes first in storage. DEC VAX and Intel x86 are examples of computers using this byte ordering.

D.2 Structure Mapping Examples

Figure 35 on page 146 shows an example of a C language structure **s** containing an assortment of scalars and one character string. The value presumed to be in each structure element is shown in hex in the C comments; these values are used below to show how the bytes making up each structure element are mapped into storage.

Note that C structure mapping rules will introduce padding (skipped bytes) in the map in order to align the scalars on their proper boundaries: 4 bytes between **a** and **b**, one byte between **d** and **e**, and two bytes between **e** and **f**. The same amount of padding will be present for both Big-Endian and Little-Endian mappings.

```
struct {
    int       a;      /*  0x11121314                  word            */
    double    b;      /*  0x2122232425262728          doubleword      */
    char *    c;      /*  0x31323334                  word            */
    char      d[7];   /*  'A', 'B', 'C', 'D', 'E', 'F', 'G'   array of bytes  */
    short     e;      /*  0x5152                      halfword        */
    int       f;      /*  0x61626364                  word            */
} s;
```

Figure 35. Example of C structure, showing values of elements

D.2.1 Big-Endian mapping

The Big-Endian mapping of structure **s** is shown in Figure 36. Addresses are shown in hex at the left of each doubleword, and in small figures below each byte. The content of each byte, as indicated in the C example in Figure 35, is shown in hex (as characters for the elements of the string).

00	11	12	13	14				
	00	01	02	03	04	05	06	07
08	21	22	23	24	25	26	27	28
	08	09	0A	0B	0C	0D	0E	0F
10	31	32	33	34	'A'	'B'	'C'	'D'
	10	11	12	13	14	15	16	17
18	'E'	'F'	'G'		51	52		
	18	19	1A	1B	1C	1D	1E	1F
20	61	62	63	64				
	20	21	22	23				

Figure 36. Big-Endian mapping of structure 's'

D.2.2 Little-Endian mapping

The same structure **s** is shown mapped Little-Endian style in Figure 37. Doublewords are shown laid out right-to-left, the common way of showing storage maps for Little-Endian machines.

				11	12	13	14	00
07	06	05	04	03	02	01	00	
21	22	23	24	25	26	27	28	08
0F	0E	0D	0C	0B	0A	09	08	
'D'	'C'	'B'	'A'	31	32	33	34	10
17	16	15	14	13	12	11	10	
		51	52		'G'	'F'	'E'	18
1F	1E	1D	1C	1B	1A	19	18	
				61	62	63	64	20
				23	22	21	20	

Figure 37. Little-Endian mapping of structure 's'

D.3 PowerPC Byte Ordering

By default, PowerPC's byte ordering is Big-Endian. Unless an overt action (described below) is taken following power-on reset, byte ordering will be as shown in Figure 36 above.

However, it is possible to run a PowerPC system in *Little-Endian mode*, such that the computational instruction set behaves as if the byte ordering were Little-Endian as in Figure 37. To do this requires setting a bit in a Special Purpose Register that controls byte ordering. Which bit is used, and which SPR contains the bit, is implementation-dependent and is specified in Book IV, *PowerPC Implementation Features* for each implementation. The symbolic name of the bit is LM, Little-Endian Mapping.

The LM bit is cleared to 0 (Big-Endian mode) on power-on reset and may be set to 1 (Little-Endian mode) or reset to 0 by a privileged ***Move To Special Purpose Register*** (***mtspr***) instruction. An implementation may require that the ***mtspr*** be accompanied by certain synchronization instructions or that a specific sequence of instructions be used to modify LM; see Book IV.

D.4 PowerPC Data Storage Addressing with LM=1

One might expect that a PowerPC system operating with LM=1 would have to perform a 2-way, 4-way, or 8-way byte swap when transferring a halfword, word, or doubleword between storage and a general or floating point register. Instead, PowerPC achieves the effect of Little-Endian byte ordering by manipulating the three low-order bits of the Effective Address (EA) as described below; no swapping of bytes is done, and individual multi-byte scalars actually appear in storage in Big-Endian byte order. The primary effect of setting LM=1 is to adjust the way Effective Addresses are computed, with the transfer of data between storage and registers unaffected and thus unencumbered by multiplexors for byte swapping.

D.4.1.1 Aligned Scalars

This discussion applies to scalar data that are aligned on their natural boundaries. For unaligned data see D.4.2, "Unaligned Scalars" on page 148; for non-scalar data see D.4.3, "Non-Scalars" on page 148. For the following ***Load*** and ***Store*** instructions the Effective Address is computed as specified in the instruction descriptions and is then modified as shown in the table below.

lbz Load Byte and Zero
lbzx Load Byte and Zero Indexed
lbzu Load Byte and Zero with Update
lbzux Load Byte and Zero with Update Indexed
lhz Load Halfword and Zero
lhzx Load Halfword and Zero Indexed
lhzu Load Halfword and Zero with Update
lhzux Load Halfword and Zero with Update Indexed
lha Load Halfword Algebraic
lhax Load Halfword Algebraic Indexed
lhau Load Halfword Algebraic with Update
lhaux Load Halfword Algebraic with Update Indexed
lhbrx Load Halfword Byte-Reverse Indexed
lwz Load Word and Zero
lwzx Load Word and Zero Indexed
lwzu Load Word and Zero with Update
lwzux Load Word and Zero with Update Indexed
lwa Load Word Algebraic
lwax Load Word Algebraic Indexed
lwaux Load Word Algebraic with Update Indexed
lwbrx Load Word Byte-Reverse Indexed
lwarx Load Word and Reserve Indexed
ld Load Doubleword
ldx Load Doubleword Indexed
ldu Load Doubleword with Update
ldux Load Doubleword with Update Indexed
ldarx Load Doubleword and Reserve Indexed
lfs Load Floating-Point Single
lfsx Load Floating-Point Single Indexed
lfsu Load Floating-Point Single with Update
lfsux Load Floating-Point Single with Update Indexed
lfd Load Floating-Point Double
lfdx Load Floating-Point Double Indexed
lfdu Load Floating-Point Double with Update
lfdux Load Floating-Point Double with Update Indexed
stb Store Byte
stbx Store Byte Indexed
stbu Store Byte with Update
stbux Store Byte with Update Indexed
sth Store Halfword
sthx Store Halfword Indexed
sthu Store Halfword with Update
sthux Store Halfword with Update Indexed
sthbrx Store Halfword Byte-Reverse Indexed
stw Store Word
stwx Store Word Indexed
stwu Store Word with Update
stwux Store Word with Update Indexed
stwbrx Store Word Byte-Reverse Indexed
stwcx. Store Word Conditional Indexed
std Store Doubleword
stdx Store Doubleword Indexed
stdu Store Doubleword with Update
stdux Store Doubleword with Update Indexed
stdcx. Store Doubleword Conditional Indexed
stfs Store Floating-Point Single
stfsx Store Floating-Point Single Indexed
stfsu Store Floating-Point Single with Update
stfsux Store Floating-Point Single with Update Indexed
stfd Store Floating-Point Double
stfdx Store Floating-Point Double Indexed
stfdu Store Floating-Point Double with Update
stfdux Store Floating-Point Double with Update Indexed
stfiwx Store Floating-Point as Integer Word Indexed

Data width (bytes)	EA modified:
8	(no change)
4	XOR with 0b100
2	XOR with 0b110
1	XOR with 0b111

The modified EA is then passed to the data cache or to main storage and the specified width of data is transferred between a general or floating-point register and the (as modified) addressed storage locations(s). The EA modification makes it appear to the processor that data is stored Little-Endian, while in fact it is stored following Big-Endian byte order *but not in the same bytes within doublewords as with LM=0.*

To continue the example of structure **s**, the structure would be placed in storage as follows, from the point of view of the cache and memory subsystem (i.e., *after* the EA modification, above):

00					11	12	13	14
	00	01	02	03	04	05	06	07
08	21	22	23	24	25	26	27	28
	08	09	0A	0B	0C	0D	0E	0F
10	'D'	'C'	'B'	'A'	31	32	33	34
	10	11	12	13	14	15	16	17
18			51	52		'G'	'F'	'E'
	18	19	1A	1B	1C	1D	1E	1F
20					61	62	63	64
	20	21	22	23	24	25	26	27

Figure 38. PowerPC Little-Endian, structure 's' in storage or cache

Because of the modifications performed on Effective Addresses, the same structure **s** *appears to the*

processor to be mapped into storage this way when LM = 1 (Little-Endian mapping):

				11	12	13	14	00
07	06	05	04	03	02	01	00	
21	22	23	24	25	26	27	28	08
0F	0E	0D	0C	0B	0A	09	08	
'D'	'C'	'B'	'A'	31	32	33	34	10
17	16	15	14	13	12	11	10	
		51	52		'G'	'F'	'E'	18
1F	1E	1D	1C	1B	1A	19	18	
				61	62	63	64	20
				23	22	21	20	

Figure 39. PowerPC Little-Endian, structure 's' as seen by processor

Note that, *as seen by the program executing in the processor*, the mapping for structure s is identical to the Little-Endian mapping shown in Figure 37. From a point of view outside the processor, however, the addresses of the bytes making up structure s are as shown in Figure 38. These addresses match neither the Big-Endian mapping of Figure 36 nor the Little-Endian mapping of Figure 37; allowance must be made for this when performing I/O in Little-Endian mode (see Section D.6).

D.4.2 Unaligned Scalars

The "trick" of exclusive-oring the low order bits of the address of a scalar does not work unless the scalar is aligned on a boundary equal to a multiple of its length. When executing in Little-Endian mode (LM = 1), PowerPC implementations may take an Alignment Interrupt (see Book III, *PowerPC Operating Environment Architecture*) whenever any of the load or store instructions listed in Section D.4.1.1 is issued with an unaligned Effective Address, regardless of whether such an access could be handled without interrupt in Big-Endian mode (LM = 0).

PowerPC systems are not *required* to take an Alignment Interrupt on unaligned accesses when LM = 1. The hardware may be designed to handle some or all such accesses just as when LM = 0. The architectural requirement is that halfwords, words, and doublewords be placed in memory such that the Little-Endian address of the lowest-order byte is the Effective Address computed by the load or store instruction, the Little-Endian address of the next-lowest-order byte is one greater, and so on. Figure 40 shows an example of a word (4 bytes) stored at Little-Endian address 5. The word is presumed to contain the binary value 0x11121314.

12	13	14						00
07	06	05	04	03	02	01	00	
							11	08
0F	0E	0D	0C	0B	0A	09	08	

Figure 40. PowerPC Little-Endian, word stored at address 5

This same word, stored by a Little-Endian program but seen from the point of view of the memory subsystem (i.e., using Big-Endian addresses), appears as shown in Figure 41:

00	12	13	14					
	00	01	02	03	04	05	06	07
08								11
	08	09	0A	0B	0C	0D	0E	0F

Figure 41. Word stored at Little-Endian address 5 as seen by Big-Endian addressing

Note that the unaligned word in this example spans two doublewords. The two parts of the unaligned word are not contiguous in Big-Endian addressing space.

An implementation may choose to support some but not all unaligned Little-Endian accesses. For example, unaligned Little-Endian accesses which are contained within a single doubleword may be supported, while those that span doublewords may trigger Alignment Interrupts.

D.4.3 Non-Scalars

PowerPC has two types of instructions that handle non-scalars, that is, multiple instances of scalars. Neither type can deal with the modified Effective Addresses required in Little-Endian mode; both types cause Alignment Interrupts (see Book III).

D.4.3.1 String Operations

The following instructions cause Alignment Interrupts when executed in Little-Endian mode (LM = 1).

lswi	Load String Word Immediate
lswx	Load String Word Indexed
stswi	Store String Word Immediate
stswx	Store String Word Indexed

String accesses are inherently unaligned; they transfer word-length quantities between storage (cache) and registers, but the quantities are not necessarily aligned on word boundaries.

Programming Note

It is up to system software to decide whether to handle the Alignment Interrupts caused by string operations in Little-Endian mode by emulating the instructions and resuming the interrupted program, or to treat the string operations as illegal and terminate the program.

As Little-Endian mode programs on PowerPC are by definition new (not old Power binaries), it is probably best not to have the compiler generate these instructions in Little-Endian mode since emulation would be slower than processing the string in-line or via subroutine call.

D.4.3.2 Load and Store Multiple

The following instructions cause Alignment Interrupts when executed in Little-Endian mode (LM=1).

lmw Load Multiple Word
stmw Store Multiple Word

While the words addressed by these instructions are on word boundaries, each word is in the opposite half of its containing doubleword from where it would be in Big-Endian mode.

Programming Note

It is up to system software to decide whether to handle the Alignment Interrupts caused by load and store multiple operations in Little-Endian mode by emulating the instructions and resuming the interrupted program, or to treat the string operations as illegal and terminate the program.

As Little-Endian mode programs on PowerPC are by definition new (not old Power binaries), it is probably best not to have the compiler generate these instructions in Little-Endian mode since emulation would be slower than a series of in-line loads and stores or a subroutine call.

D.5 PowerPC Instruction Storage Addressing with LM=1

Each PowerPC instruction occupies 32 bits (one word) of storage. PowerPC fetches and executes instructions as if the Current Instruction Address (CIA) had been advanced one word for each sequential instruction. When operating with LM=1, the CIA is modified according to the Little Endian rule for fetching word-length scalars: it is exclusive-ORed with 0b100. A program is thus an array of Little-Endian words with each word fetched and executed in order (discounting branches).

As an example, consider the following fragment of assembly-language code:

```
loop:
     cmplwi    r5, 0
     beq       done
     lwzux     r4, r5, r6
     add       r7, r7, r4
     subi      r5, 1
     b         loop
done:
     stw       r7, total
```

These instructions are mapped into storage for Big-Endian execution in the as shown in Figure 42 (assume the program starts at address 0).

00	loop: cmplwi r5,0				beq done			
	00	01	02	03	04	05	06	07
08	lwzux r4,r5,r6				add r7,r7,r4			
	08	09	0A	0B	0C	0D	0E	0F
10	subi r5,1				b loop			
	10	11	12	13	14	15	16	17
18	done: stw r7,total							
	18	19	1A	1B	1C	1D	1E	1F

Figure 42. PowerPC Big-Endian, instruction sequence as seen by processor

If this same program is assembled for and executed in Little-Endian mode, the mapping seen by the processor appears as shown in Figure 43.

beq done				loop: cmplwi				00
07	06	05	04	03	02	01	00	
add r7,r7,r4				lwzux r4,r5,r6				08
0F	0E	0D	0C	0B	0A	09	08	
b loop				subi r5,1				10
17	16	15	14	13	12	11	10	
				done: stw r7,total				18
1F	1E	1D	1C	1B	1A	19	18	

Figure 43. PowerPC Little-Endian, instruction sequence as seen by processor

Each machine instruction appears in storage as a 32-bit integer containing the value described in the instruction description, regardless of whether LM=0 or LM=1. This is a consequence of the fact that scalars are always mapped in storage in Big-Endian byte order.

When LM=1 (Little-Endian mapping), *all* references to the instruction stream must follow Little-Endian addressing, including addresses saved in system registers on interrupt, return addresses saved in the Link Register, and branch displacements and addresses.

- An instruction address placed in the Link Register by ***Branch and Link*** or an instruction address saved in a Special Purpose Register on interrupt

must be the address that a program executing in Little-Endian mode would use to access the instruction as a word of data using a load instruction.

- An offset in a relative branch instruction must reflect the difference between the addresses of the instructions, where the addresses used are those that a program executing in Little-Endian mode would use to access the instructions as data words using a load instruction.
- A target address in an absolute branch instruction must be the address that a program executing in Little-Endian mode would use to access the target instruction as a word of data using a load instruction.

D.6 PowerPC Input/Output with LM=1

Input/output, such as writing the contents of a storage page to disk, transfers a *byte stream* on both Big-Endian and Little-Endian systems. For the disk transfer, byte 0 of the page is written to the first byte of the disk record and so on.

For a PowerPC system running in Big-Endian mode, I/O transfers happen "naturally" because the byte that the processor sees as byte 0 is the same one that the storage subsystem sees as byte 0.

For a PowerPC system running in Little-Endian mode, this is not the case because of the modification of the three low-order bits of the Effective Address when the processor accesses storage. In order for I/O transfers to give the appearance of transferring byte streams properly, **in Little-Endian mode (LM=1) I/O transfers must be performed as if the bytes transferred were accessed one byte at a time, using the Little-Endian address modification appropriate for single-byte transfers (exclusive-or with 0b111).** This does not mean that I/O on Little-Endian PowerPC machines must be done using only 1-byte-wide transfers; data transfers can be as wide as desired, but the order of the bytes transferred within doublewords must be as if the bytes were fetched or stored one at a time.

System Architecture Note

It is beyond the scope of the PowerPC Architecture to specify how such byte ordering is done in the I/O path to memory. System architecture must provide a means for this to be done in a system that is to be run in Little-Endian mode.

Note that not all I/O done on PowerPC systems is for large blocks as described above. I/O can be performed with certain devices by merely storing to or loading from addresses that are associated with the devices (the terms "memory-mapped I/O" and "programmed I/O" or "PIO" are used for this). For such PIO transfers, care must be taken when defining the addresses to be used, for these addresses will be subjected to the Effective Address modifications shown in the table in D.4.1.1, "Aligned Scalars" on page 147. A load or store that maps to a control register on a device may require that the value transferred have its bytes reversed; if this is required, the loads and stores described in 3.3.4, "Fixed-Point Load and Store with Byte Reversal Instructions" on page 40 may be used. Note that any requirement for such byte reversal for a particular I/O device register is independent of whether PowerPC is running in Big-Endian or Little-Endian mode.

D.7 Origin of Endian

The terms *Big-Endian* and *Little-Endian* come from Part I, Chapter 4, of Jonathan Swift's *Gulliver's Travels*. Here is the complete passage, from the 1734 edition.

> Our Histories of six Thousand Moons make no Mention of any other Regions, than the two great Empires of *Lilliput* and *Blefuscu*. Which two mighty Powers have, as I was going to tell you, been engaged in a most obstinate War for six and thirty Moons past. It began upon the following Occasion. It is allowed on all Hands, that the primitive Way of breaking Eggs before we eat them, was upon the larger End: But his present Majesty's Grand-father, while he was a Boy, going to eat an Egg, and breaking it according to the ancient Practice, happened to cut one of his Fingers. Whereupon the Emperor his Father, published an Edict, commanding all his Subjects, upon great Penalties, to break the smaller End of their Eggs. The People so highly resented this Law, that our Histories tell us, there have been six Rebellions raised on that Account; wherein one Emperor lost his Life, and another his Crown. These civil Commotions were constantly fomented by the Monarchs of *Blefuscu*; and when they were quelled, the Exiles always fled for Refuge to that Empire. It is computed that eleven Thousand Persons have, at several Times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large Volumes have been published upon this Controversy: But the Books of the *Big-Endians* have been long forbidden, and the whole Party rendered incapable by Law of holding Employments. During the Course of these Troubles, the Emperors of *Blefuscu* did frequently expostulate by their Ambassadors, accusing us of making a Schism in Religion, by offending against a fundamental Doctrine of our great Prophet *Lustrog*, in the fifty-

fourth Chapter of the *Brundrecal*, (which is their *Alcoran*.) This, however, is thought to be a mere Strain upon the text: For the Words are these; *That all true Believers shall break their Eggs at the convenient End*: and which is the convenient End, seems, in my humble Opinion, to be left to every Man's Conscience, or at least in the Power of the chief Magistrate to determine. Now the *Big-Endian* Exiles have found so much Credit in the Emperor of *Blefuscu's* Court; and so much private Assistance and Encouragement from their Party here at home, that a bloody War has been carried on between the two Empires for six and thirty Moons with various Success; during which Time we have lost Forty Capital Ships, and a much greater Number of smaller Vessels, together with thirty thousand of our best Seamen and Soldiers; and the Damage received by the Enemy is reckoned to be somewhat greater than ours. However, they have now equipped a numerous Fleet, and are just preparing to make a Descent upon us: and his Imperial Majesty, placing great Confidence in your Valour and Strength, hath commanded me to lay this Account of his Affairs before you.

Appendix E. Programming Examples

E.1 Synchronization

This appendix gives examples of how the *Synchronization* instructions can be used to emulate various synchronization primitives, and to provide more complex forms of synchronization.

For each of these examples, it is assumed that a similar sequence of instructions is used by all processes requiring synchronization on the accessed data.

The examples deal with words: they can be used for doublewords by changing all ***lwarx*** instructions to ***ldarx***, all ***stwcx.*** instructions to ***stdcx.***, all ***stw*** instructions to ***std***, and all ***cmpw[i]*** extended mnemonics to ***cmpd[i]***.

E.1.1 Synchronization Primitives

The following examples show how the ***lwarx*** and ***stwcx.*** instructions can be used to emulate various synchronization primitives.

The sequences used to emulate the various primitives consist primarily of a loop using ***lwarx*** and ***stwcx.***. No additional synchronization is necessary, because the ***stwcx.*** will fail, setting the EQ bit to 0, if the word loaded by ***lwarx*** has changed before the ***stwcx.*** is executed: see Book II, *PowerPC Virtual Environment Architecture* for more detail.

Fetch and No-op

The "Fetch and No-op" primitive atomically loads the current value in a word in storage.

In this example it is assumed that the address of the word to be loaded is in GPR 3 and the data loaded are returned in GPR 4.

```
loop: lwarx  r4,0,r3      #load and reserve
      stwcx. r4,0,r3      #store old value if
                          #  still reserved
      bne    loop         #loop if lost reserv'n
```

Notes:

1. Because ***stwcx.*** is not necessarily performed with respect to all other mechanisms that access storage (see Book II, *PowerPC Virtual Environment Architecture*), an ordinary *Load* instruction, or even a *Load and Reserve* instruction, on a different processor, may return a "stale" value. However, a subsequent ***lwarx*** on the other processor followed by a successful ***stwcx.*** on that processor is guaranteed to have returned the value stored by the first processor's ***stwcx.*** (in the absence of other stores to the location).
2. The storing done by the ***stwcx.*** instruction in this example is redundant.

Fetch and Store

The "Fetch and Store" primitive atomically loads and replaces a word in storage.

In this example it is assumed that the address of the word to be loaded and replaced is in GPR 3, the new value is in GPR 4, and the old value is returned in GPR 5.

```
loop: lwarx  r5,0,r3      #load and reserve
      stwcx. r4,0,r3      #store new value if
                          #  still reserved
      bne    loop         #loop if lost reserv'n
```

Fetch and Add

The "Fetch and Add" primitive atomically increments a word in storage.

In this example it is assumed that the address of the word to be incremented is in GPR 3, the increment is in GPR 4, and the old value is returned in GPR 5.

```
loop: lwarx  r5,0,r3      #load and reserve
      add    r0,r4,r5     #increment word
      stwcx. r0,0,r3      #store new value if
                          #  still reserved
      bne    loop         #loop if lost reserv'n
```

Fetch and AND

The "Fetch and AND" primitive atomically ANDs a value into a word in storage.

In this example it is assumed that the address of the word to be ANDed is in GPR 3, the value to AND into it is in GPR 4, and the old value is returned in GPR 5.

```
loop: lwarx  r5,0,r3      #load and reserve
      and    r0,r4,r5     #AND word
      stwcx. r0,0,r3      #store new value if
                          #  still reserved
      bne    loop         #loop if lost reserv'n
```

Notes:

1. The sequence given above can be changed to perform another Boolean operation atomically on a word in storage, simply by changing the ***and*** instruction to the desired Boolean instruction (***or, xor,*** etc.).

Test and Set

The "Test and Set" primitive atomically loads a word from storage, ensures that the word in storage contains a non-zero value, and sets the EQ bit of CR Field 0 according to whether the value loaded is zero.

In this example it is assumed that the address of the word to be tested is in GPR 3, the new value (non-zero) is in GPR 4, and the old value is returned in GPR 5.

```
loop: lwarx  r5,0,r3      #load and reserve
      cmpwi  r5,0         #done if word
      bne    $+12         #  not equal to 0
      stwcx. r4,0,r3      #try to store non-0
      bne    loop         #loop if lost reserv'n
```

Notes:

1. "Test and Set" is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by ***lwarx*** and ***stwcx.***. A major weakness of "Test and Set" is that it does not scale well. Using "Test and Set" before a "critical section" allows at most one process to execute in the critical section at a time. Using ***lwarx*** and ***stwcx.*** to bracket the critical section allows many processes to execute in the critical section at once, but at most one will succeed in exiting from the section with its results stored.
2. Depending on the application, if *Test and Set* fails (i.e., sets the EQ bit of CR Field 0 to zero) it may be appropriate to re-execute the *Test and Set*.

Compare and Swap

The "Compare and Swap" primitive atomically compares a value in a register with a word in storage, if they are surely equal stores the value from a second register into the word in storage, if they may be unequal loads the word from storage into the first register, and sets the EQ bit of CR Field 0 to indicate the result of the comparison.

In this example it is assumed that the address of the word to be tested is in GPR 3, the comparand is in GPR 4, the new value is in GPR 5, and the old value is returned in GPR 6.

```
      lwarx  r6,0,r3      #load and reserve
      cmpw   r4,r6        #1st 2 operands equal?
      bne    $+8          #skip if not
      stwcx. r5,0,r3      #store new value if
                          #  still reserved
```

Notes:

1. "Compare and Swap" is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by ***lwarx*** and ***stwcx.***. A major weakness of typical "Compare and Swap" instructions is that they permit spurious success if the word being tested has changed and then changed back to its old value: the sequence shown above does not have this weakness.
2. Depending on the application, if *Compare and Swap* fails (i.e., sets the EQ bit of CR Field 0 to zero) it may be appropriate to recompute the value potentially to be stored and then re-execute the *Compare and Swap*.

E.1.2 List Insertion

The following example shows how the ***lwarx*** and ***stwcx.*** instructions can be used to implement simple LIFO (last in first out) insertion into a singly linked list. (Complicated list insertion, in which multiple values must be changed atomically, or in which the correct order of insertion depends on the contents of the elements, cannot be implemented in the manner shown below, and requires a more complicated strategy such as using locks.)

The "next element pointer" from the list element after which the new element is to be inserted, here called the "parent element," is stored into the new element, so that the new element points to the next element in the list: this store is performed unconditionally. Then the address of the new element is conditionally stored

into the parent element, thereby adding the new element to the list.

In this example it is assumed that the address of the parent element is in GPR 3, the address of the new element is in GPR 4, and the next element pointer is at offset 0 from the start of the element. It is also assumed that the next element pointer of each list element is in a "reservation granule" separate from that of the next element pointer of all other list elements: see Book II, *PowerPC Virtual Environment Architecture*.

```
loop: lwarx  r2,0,r3    #get next pointer
      stw    r2,0(r4)   #store in new element
      sync              #let store settle (can
                        # omit if not MP)
      stwcx. r4,0,r3    #add new element to list
      bne    loop       #loop if stwcx. failed
```

In the preceding example, if two list elements have next element pointers in the same reservation granule then, in a multiprocessor, "livelock" can occur. (Livelock is a state in which processors interact in a way such that no processor makes progress.)

If it is not possible to allocate list elements such that each element's next element pointer is in a different reservation granule, then livelock can be avoided by using the following, more complicated, code sequence.

```
      lwz    r2,0(r3)   #get next pointer
loop1: mr    r5,r2      #keep a copy
      stw    r2,0(r4)   #store in new element
      sync              #let store settle
loop2: lwarx r2,0,r3    #get it again
      cmpw   r2,r5      #loop if changed (someone
      bne    loop1      # else progressed)
      stwcx. r4,0,r3    #add new element to list
      bne    loop2      #loop if failed
```

E.1.3 Notes

1. In general, ***lwarx*** and ***stwcx.*** instructions should be paired, with the same effective address used for both. The exception is an isolated ***stwcx.*** instruction that is used to clear any existing reservation on the processor, for which there is no paired ***lwarx*** and for which any (scratch) effective address can be used.
2. It is acceptable to execute a ***lwarx*** instruction for which no ***stwcx.*** instruction is executed. For example, such a "dangling ***lwarx***" occurs if the value loaded in the "Test and Set" sequence shown above is not zero.
3. To increase the likelihood that forward progress is made, it is important that looping on ***lwarx/stwcx.*** pairs be minimized. For example, in the sequence shown above for "Test and Set," this is achieved by testing the old value before attempting the store: were the order reversed, more ***stwcx.*** instructions might be executed, and reservations might more often be lost between the ***lwarx*** and the ***stwcx.***.
4. The manner in which ***lwarx*** and ***stwcx.*** are communicated to other processors and mechanisms, and between levels of the storage subsystem within a given processor (see Book II, *PowerPC Virtual Environment Architecture*), is implementation-dependent. In some implementations performance may be improved by minimizing looping on a ***lwarx*** instruction that fails to return a desired value. For example, in the "Test and Set" example shown above, if the programmer wishes to stay in the loop until the word loaded is zero, he could change the "bne $+12" to "bne loop." However, in some implementations better performance may be obtained by using an ordinary *Load* instruction to do the initial checking of the value, as follows.

   ```
   loop: lwz    r5,0(r3)   #load the word
         cmpwi  r5,0       #loop back if word
         bne    loop       #  not equal to 0
         lwarx  r5,0,r3    #try again, reserving
         cmpwi  r5,0       #  (likely to succeed)
         bne    loop
         stwcx. r4,0,r3    #try to store non-0
         bne    loop       #loop if lost reserv'n
   ```

5. In a multiprocessor, livelock is possible if a loop containing a ***lwarx/stwcx.*** pair also contains an ordinary *Store* instruction for which any byte of the affected storage area is in the reservation granule of the reservation: see Book II, *PowerPC Virtual Environment Architecture*. For example, the first code sequence shown in Section E.1.2, List Insertion, can cause livelock if two list elements have next element pointers in the same reservation granule.

E.2 Multiple-Precision Shifts

This appendix gives examples of how multiple-precision shifts can be programmed.

A multiple-precision shift is initially defined to be a shift of an N-doubleword quantity (64-bit mode) or an N-word quantity (32-bit mode), where N>1. (This definition is relaxed somewhat for 32-bit mode, below.) The quantity to be shifted is contained in N registers (in the low-order 32 bits in 32-bit mode). The shift amount is specified either by an immediate value in the instruction, or by bits 57:63 (64-bit mode) or 58:63 (32-bit mode) of a register.

The examples shown below distinguish between the cases N=2 and N>2. If N=2, the shift amount may be in the range 0 through 127 (64-bit mode) or 0 through 63 (32-bit mode), which are the maximum ranges supported by the *Shift* instructions used. However if N>2, the shift amount must be in the range 0 through 63 (64-bit mode) or 0 through 31 (32-bit mode), in order for the examples to yield the desired result. The specific instance shown for N>2 is N=3: extending those instruction sequences to larger N is straightforward, as is reducing them to the case N=2 when the more stringent restriction on shift amount is met. For shifts with immediate shift amounts only the case N=3 is shown, because the more stringent restriction on shift amount is always met.

In the examples it is assumed that GPRs 2 and 3 (and 4) contain the quantity to be shifted, and that the result is to be placed into the same registers, except for the immediate left shifts in 64-bit mode for which the result is placed into GPRs 3, 4, and 5. In all cases, for both input and result, the lowest-numbered register contains the highest-order part of the data and highest-numbered register contains the lowest-order part. In 32-bit mode, the high-order 32 bits of these registers are assumed not to be part of the quantity to be shifted nor of the result. For non-immediate shifts, the shift amount is assumed to be in bits 57:63 (64-bit mode) or 58:63 (32-bit mode) of GPR 6. For immediate shifts, the shift amount is assumed to be greater than 0. GPRs 0 and 31 are used as scratch registers.

For N>2, the number of instructions required is 2N−1 (immediate shifts) or 3N−1 (non-immediate shifts).

Multiple-precision shifts in 64-bit mode

Shift Left Immediate, N = 3 (shift amnt < 64)

```
rldicr    r5,r4,sh,63-sh
rldimi    r4,r3,0,sh
rldicl    r4,r4,sh,0
rldimi    r3,r2,0,sh
rldicl    r3,r3,sh,0
```

Shift Left, N = 2 (shift amnt < 128)

```
subfic    r31,r6,64
sld       r2,r2,r6
srd       r0,r3,r31
or        r2,r2,r0
addic     r31,r6,-64
sld       r0,r3,r31
or        r2,r2,r0
sld       r3,r3,r6
```

Multiple-precision shifts in 32-bit mode

Shift Left Immediate, N = 3 (shift amnt < 32)

```
rlwinm    r2,r2,sh,0,31-sh
rlwimi    r2,r3,sh,32-sh,31
rlwinm    r3,r3,sh,0,31-sh
rlwimi    r3,r4,sh,32-sh,31
rlwinm    r4,r4,sh,0,31-sh
```

Shift Left, N = 2 (shift amnt < 64)

```
subfic    r31,r6,32
slw       r2,r2,r6
srw       r0,r3,r31
or        r2,r2,r0
addic     r31,r6,-32
slw       r0,r3,r31
or        r2,r2,r0
slw       r3,r3,r6
```

Multiple-precision shifts in 64-bit mode, continued

Shift Left, N = 3 (shift amnt < 64)

```
subfic    r31,r6,64
sld       r2,r2,r6
srd       r0,r3,r31
or        r2,r2,r0
sld       r3,r3,r6
srd       r0,r4,r31
or        r3,r3,r0
sld       r4,r4,r6
```

Shift Right Immediate, N = 3 (shift amnt < 64)

```
rldimi    r4,r3,0,64-sh
rldicl    r4,r4,64-sh,0
rldimi    r3,r2,0,64-sh
rldicl    r3,r3,64-sh,0
rldicl    r2,r2,64-sh,sh
```

Shift Right, N = 2 (shift amnt < 128)

```
subfic    r31,r6,64
srd       r3,r3,r6
sld       r0,r2,r31
or        r3,r3,r0
addic     r31,r6,-64
srd       r0,r2,r31
or        r3,r3,r0
srd       r2,r2,r6
```

Shift Right, N = 3 (shift amnt < 64)

```
subfic    r31,r6,64
srd       r4,r4,r6
sld       r0,r3,r31
or        r4,r4,r0
srd       r3,r3,r6
sld       r0,r2,r31
or        r3,r3,r0
srd       r2,r2,r6
```

Shift Right Algebraic Immediate, N = 3 (shift amnt < 64)

```
rldimi    r4,r3,0,64-sh
rldicl    r4,r4,64-sh,0
rldimi    r3,r2,0,64-sh
rldicl    r3,r3,64-sh,0
sradi     r2,r2,sh
```

Shift Right Algebraic, N = 2 (shift amnt < 128)

```
subfic    r31,r6,64
srd       r3,r3,r6
sld       r0,r2,r31
or        r3,r3,r0
addic.    r31,r6,-64
srad      r0,r2,r31
ble       $+8
ori       r3,r0,0
srad      r2,r2,r6
```

Multiple-precision shifts in 32-bit mode, continued

Shift Left, N = 3 (shift amnt < 32)

```
subfic    r31,r6,32
slw       r2,r2,r6
srw       r0,r3,r31
or        r2,r2,r0
slw       r3,r3,r6
srw       r0,r4,r31
or        r3,r3,r0
slw       r4,r4,r6
```

Shift Right Immediate, N = 3 (shift amnt < 32)

```
rlwinm    r4,r4,32-sh,sh,31
rlwimi    r4,r3,32-sh,0,sh-1
rlwinm    r3,r3,32-sh,sh,31
rlwimi    r3,r2,32-sh,0,sh-1
rlwinm    r2,r2,32-sh,sh,31
```

Shift Right, N = 2 (shift amnt < 64)

```
subfic    r31,r6,32
srw       r3,r3,r6
slw       r0,r2,r31
or        r3,r3,r0
addic     r31,r6,-32
srw       r0,r2,r31
or        r3,r3,r0
srw       r2,r2,r6
```

Shift Right, N = 3 (shift amnt < 32)

```
subfic    r31,r6,32
srw       r4,r4,r6
slw       r0,r3,r31
or        r4,r4,r0
srw       r3,r3,r6
slw       r0,r2,r31
or        r3,r3,r0
srw       r2,r2,r6
```

Shift Right Algebraic Immediate, N = 3 (shift amnt < 32)

```
rlwinm    r4,r4,32-sh,sh,31
rlwimi    r4,r3,32-sh,0,sh-1
rlwinm    r3,r3,32-sh,sh,31
rlwimi    r3,r2,32-sh,0,sh-1
srawi     r2,r2,sh
```

Shift Right Algebraic, N = 2 (shift amnt < 64)

```
subfic    r31,r6,32
srw       r3,r3,r6
slw       r0,r2,r31
or        r3,r3,r0
addic.    r31,r6,-32
sraw      r0,r2,r31
ble       $+8
ori       r3,r0,0
sraw      r2,r2,r6
```

Multiple-precision shifts in 64-bit mode, continued

Shift Right Algebraic, N = 3 (shift amnt < 64)

```
subfic    r31,r6,64
srd       r4,r4,r6
sld       r0,r3,r31
or        r4,r4,r0
srd       r3,r3,r6
sld       r0,r2,r31
or        r3,r3,r0
srad      r2,r2,r6
```

Multiple-precision shifts in 32-bit mode, continued

Shift Right Algebraic, N = 3 (shift amnt < 32)

```
subfic    r31,r6,32
srw       r4,r4,r6
slw       r0,r3,r31
or        r4,r4,r0
srw       r3,r3,r6
slw       r0,r2,r31
or        r3,r3,r0
sraw      r2,r2,r6
```

The examples shown above for 32-bit mode work both in 32-bit mode of a 64-bit implementation and in a 32-bit implementation. They perform the shift in units of words. If ability to run in 32-bit implementations is not required, in a 64-bit implementation better performance can be obtained in 32-bit mode than that of the examples shown above, by using all 64 bits of GPRs 2 and 3 (and 4) to contain the quantity to be shifted, and placing the result into all 64 bits of the same registers.

Let N be the number of doublewords to be shifted.

The examples shown above for 64-bit mode work equally well in 32-bit mode of a 64-bit implementation, using all 64 bits of the registers. For N>2, the number of instructions required is 2N−1 (immediate shifts) or 3N−1 (non-immediate shifts), compared with 4N−1 (immediate shifts) or 6N−1 (non-immediate shifts) for the examples shown above for 32-bit mode. (The examples shown above require using twice as many registers to hold the quantity to be shifted.)

E.3 Floating-Point Conversions

This appendix gives examples of how the *Floating-Point Conversion* instructions can be used to perform various conversions.

Warning: Some of the examples use the ***fsel*** instruction. Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities: see Section E.4.4, "Notes" on page 162.

E.3.1 Conversion from Floating-Point Number to Floating-Point Integer

In a 64-bit Implementation

The full *convert to floating-point integer* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, and the result is returned in FPR 3.

```
mtfsb0   23          #clear VXCVI
fctid[z] f3,f1       #convert to fx int
fcfid    f3,f3       #convert back again
mcrfs    7,5         #VXCVI to CR
bf       31,$+8      #skip if VXCVI was 0
fmr      f3,f1       #input was fp int
```

In a 32-bit Implementation

Editors' Note

To be supplied.

E.3.2 Conversion from Floating-Point Number to Signed Fixed-Point Integer Doubleword

This example applies to 64-bit implementations only.

The full *convert to signed fixed-point integer doubleword* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the result is returned in GPR 3, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space.

```
fctid[z] f2,f1          #convert to dword int
stfd     f2,disp(r1)    #store float
ld       r3,disp(r1)    #load dword
```

E.3.3 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Doubleword

This example applies to 64-bit implementations only.

The full *convert to unsigned fixed-point integer doubleword* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the value 0 is in FPR 0, the value $2^{64}-2048$ is in FPR 3, the value 2^{63} is in FPR 4 and GPR 4, the result is returned in GPR 3, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space.

```
fsel     f2,f1,f1,f0   #use 0 if < 0
fsub     f5,f3,f1      #use max if > max
fsel     f2,f5,f2,f3
fsub     f5,f2,f4      #subtract 2**63
fcmpu    cr2,f2,f4     #use diff if ≥ 2**63
fsel     f2,f5,f5,f2
fctid[z] f2,f2         #convert to fx int
stfd     f2,disp(r1)   #store float
ld       r3,disp(r1)   #load dword
blt      cr2,$+8       #add 2**63 if input
add      r3,r3,r4      #  was ≥ 2**63
```

E.3.4 Conversion from Floating-Point Number to Signed Fixed-Point Integer Word

The full *convert to signed fixed-point integer word* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the result is returned in GPR 3, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space. The last instruction is needed only if a 64-bit result is required, and applies to 64-bit implementations only.

```
fctiw[z] f2,f1          #convert to fx int
stfd     f2,disp(r1)    #store float
lwz      r3,disp+4(r1)  #load word and zero
extsw    r3,r3          #(for 64-bit result)
```

E.3.5 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Word

In a 64-bit Implementation

The full *convert to unsigned fixed-point integer word* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the value 0 is in FPR 0, the value $2^{32}-1$ is in FPR 3, the result is returned in GPR 3, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space.

```
fsel     f2,f1,f1,f0    #use 0 if < 0
fsub     f4,f3,f1       #use max if > max
fsel     f2,f4,f2,f3
fctid[z] f2,f2          #convert to fx int
stfd     f2,disp(r1)    #store float
lwz      r3,disp+4(r1) #load word and zero
```

In a 32-bit Implementation

The full *convert to unsigned fixed-point integer word* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the value 0 is in FPR 0, the value 2^{32} is in FPR 3, the value 2^{31} is in FPR 4 and GPR 4, the result is returned in GPR 3, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space.

```
fsel     f2,f1,f1,f0    #use 0 if < 0
fsub     f5,f3,f1       #use max if > max
fsel     f2,f5,f2,f3
fsub     f5,f2,f4       #subtract 2**31
fcmpu    cr2,f2,f4      #use diff if ≥ 2**31
fsel     f2,f5,f5,f2
fctiw[z] f2,f2          #convert to fx int
stfd     f2,disp(r1)    #store float
lwz      r3,disp+4(r1) #load word
blt      cr2,$+8        #add 2**31 if input
add      r3,r3,r4       #  was ≥ 2**31
```

E.3.6 Conversion from Signed Fixed-Point Integer Doubleword to Floating-Point Number

This example applies to 64-bit implementations only.

The full *convert from signed fixed-point integer doubleword* function, using the rounding mode specified by $FPSCR_{RN}$, can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the result is returned in FPR 1, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space.

```
std      r3,disp(r1)    #store dword
lfd      f1,disp(r1)    #load float
fcfid    f1,f1          #convert to fpu int
```

E.3.7 Conversion from Unsigned Fixed-Point Integer Doubleword to Floating-Point Number

This example applies to 64-bit implementations only.

The full *convert from unsigned fixed-point integer doubleword* function, using the rounding mode specified by $FPSCR_{RN}$, can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the value 2^{32} is in FPR 4, the result is returned in FPR 1, and two doublewords at displacement "disp" from the address in GPR 1 can be used as scratch space.

```
rldicl   r2,r3,32,32    #isolate high half
rldicl   r0,r3,0,32     #isolate low half
std      r2,disp(r1)    #store dword both
std      r0,disp+8(r1)
lfd      f2,disp(r1)    #load float both
lfd      f1,disp+8(r1) #load float both
fcfid    f2,f2          #convert each half to
fcfid    f1,f1          #  fpu int (no rnd)
fmadd    f1,f4,f2,f1    #(2**32)*high + low
                        #  (only add can rnd)
```

An alternative, shorter, sequence can be used if rounding according to $FSCPR_{RN}$ is desired and $FPSCR_{RN}$ specifies *Round toward +Infinity* or *Round toward −Infinity*, or if it is acceptable for a rounded answer to be either of the two representable floating-point integers nearest algebraically to the given fixed-point integer. In this case the full *convert from unsigned fixed-point integer doubleword* function can be implemented with the sequence shown below, assuming the value 2^{64} is in FPR 2.

```
std      r3,disp(r1)    #store dword
lfd      f1,disp(r1)    #load float
fcfid    f1,f1          #convert to fpu int
fadd     f4,f1,f2       #add 2**64
fsel     f1,f1,f1,f4    #  if r3 < 0
```

E.3.8 Conversion from Signed Fixed-Point Integer Word to Floating-Point Number

In a 64-bit Implementation

The full *convert from signed fixed-point integer word* function can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the result is returned in FPR 1, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space. (Rounding cannot occur.)

```
extsw    r3,r3          #extend sign
std      r3,disp(r1)    #store dword
lfd      f1,disp(r1)    #load float
fcfid    f1,f1          #convert to fpu int
```

In a 32-bit Implementation

Editors' Note

To be supplied.

E.3.9 Conversion from Unsigned Fixed-Point Integer Word to Floating-Point Number

In a 64-bit Implementation

The full *convert from unsigned fixed-point integer word* function can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the result is returned in FPR 1, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space. (Rounding cannot occur.)

```
rldicl   r0,r3,0,32     #zero-extend
std      r0,disp(r1)    #store dword
lfd      f1,disp(r1)    #load float
fcfid    f1,f1          #convert to fpu int
```

In a 32-bit Implementation

Editors' Note

To be supplied.

E.4 Floating-Point Selection

This appendix gives examples of how the *Floating Select* instruction can be used to implement floating-point minimum and maximum functions, and certain simple forms of if-then-else constructions, without branching.

The examples show program fragments in an imaginary, C-like, high-level programming language, and the corresponding program fragment using ***fsel*** and other PowerPC instructions. In the examples, *a*, *b*, *x*, *y*, and *z* are floating-point variables, which are assumed to be in FPRs *fa*, *fb*, *fx*, *fy*, and *fz*. FPR *fs* is assumed to be available for scratch space.

Additional examples can be found in Section E.3, "Floating-Point Conversions" on page 159.

Warning: Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities: see Section E.4.4, "Notes."

E.4.1 Comparison to Zero

High-level language:	PowerPC:		Notes
`if a ≥ 0.0 then x ← y`	`fsel`	`fx,fa,fy,fz`	(1)
`else x ← z`			
`if a > 0.0 then x ← y`	`fneg`	`fs,fa`	(1,2)
`else x ← z`	`fsel`	`fx,fs,fz,fy`	
`if a = 0.0 then x ← y`	`fsel`	`fx,fa,fy,fz`	(1)
`else x ← z`	`fneg`	`fs,fa`	
	`fsel`	`fx,fs,fx,fz`	

E.4.2 Minimum and Maximum

High-level language:	PowerPC:		Notes
`x ← min(a,b)`	`fsub`	`fs,fa,fb`	(3,4,5)
	`fsel`	`fx,fs,fb,fa`	
`x ← max(a,b)`	`fsub`	`fs,fa,fb`	(3,4,5)
	`fsel`	`fx,fs,fa,fb`	

E.4.3 Simple if-then-else Constructions

High-level language:	PowerPC:		Notes
`if a ≥ b then x ← y`	`fsub`	`fs,fa,fb`	(4,5)
`else x ← z`	`fsel`	`fx,fs,fy,fz`	
`if a > b then x ← y`	`fsub`	`fs,fb,fa`	(3,4,5)
`else x ← z`	`fsel`	`fx,fs,fz,fy`	
`if a = b then x ← y`	`fsub`	`fs,fa,fb`	(4,5)
`else x ← z`	`fsel`	`fx,fs,fy,fz`	
	`fneg`	`fs,fs`	
	`fsel`	`fx,fs,fx,fz`	

E.4.4 Notes

The following Notes apply to the preceding examples, and to the corresponding cases using the other three arithmetic relations (<, ≤, and ≠). They should also be considered when any other use of ***fsel*** is contemplated.

In these Notes, the "optimized program" is the PowerPC program shown, and the "unoptimized program" is the corresponding PowerPC program that uses ***fcmpu*** and *Branch Conditional* instructions instead of ***fsel***.

1. The unoptimized program affects the VXSNAN bit of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exception is enabled, while the optimized program does not affect this bit. This is incompatible with the IEEE standard.
2. The optimized program gives the incorrect result if *a* is a NaN.
3. The optimized program gives the incorrect result if *a* and/or *b* is a NaN (except that it may give the correct result in some cases for the minimum and maximum functions, depending on how those functions are defined to operate on NaNs).
4. The optimized program gives the incorrect result if *a* and *b* are infinities of the same sign. (Here it is assumed that Invalid Operation Exceptions are disabled, in which case the result of the subtraction is a NaN. The analysis is more complicated if Invalid Operation Exceptions are enabled, because in that case the target register of the subtraction is unchanged.)
5. The optimized program affects the OX, UX, XX, and VXISI bits of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exceptions are enabled, while the unoptimized program does not affect these bits. This is incompatible with the IEEE standard.

Appendix F. Cross-Reference for Changed Power Mnemonics

The following table lists the Power instruction mnemonics that have been changed in the PowerPC Architecture, sorted by Power mnemonic.

To determine the PowerPC mnemonic for one of these Power mnemonics, find the Power mnemonic in the second column of the table: the remainder of the line gives the PowerPC mnemonic and the page or Book in which the instruction is described, as well as the instruction names. A page number is shown for instructions that are defined in this Book (Book I, *PowerPC User Instruction Set Architecture*), and the Book number is shown for instructions that are defined in other Books (Book II, *PowerPC Virtual Environment Architecture*, and Book III, *PowerPC Operating Environment Architecture*). If an instruction is defined in more than one Book, the lowest-numbered Book is used.

Power mnemonics that have not changed are not listed. Power instruction names that are the same in PowerPC are not repeated: i.e., for these, the last column of the table is blank.

Page / Bk	Power		PowerPC	
	Mnemonic	Instruction	Mnemonic	Instruction
52	a[o][.]	Add	addc[o][.]	Add Carrying
53	ae[o][.]	Add Extended	adde[o][.]	
51	ai	Add Immediate	addic	Add Immediate Carrying
51	ai.	Add Immediate and Record	addic.	Add Immediate Carrying and Record
53	ame[o][.]	Add To Minus One Extended	addme[o][.]	
63	andil.	AND Immediate Lower	andi.	AND Immediate
63	andiu.	AND Immediate Upper	andis.	AND Immediate Shifted
54	aze[o][.]	Add To Zero Extended	addze[o][.]	
21	bcc[l]	Branch Conditional to Count Register	bcctr[l]	
21	bcr[l]	Branch Conditional to Link Register	bclr[l]	
50	cal	Compute Address Lower	addi	Add Immediate
50	cau	Compute Address Upper	addis	Add Immediate Shifted
51	cax[o][.]	Compute Address	add[o][.]	Add
68	cntlz[.]	Count Leading Zeros	cntlzw[.]	Count Leading Zeros Word
Bk II	dclz	Data Cache Line Set to Zero	dcbz	Data Cache Block set to Zero
48	dcs	Data Cache Synchronize	sync	Synchronize
67	exts[.]	Extend Sign	extsh[.]	Extend Sign Halfword
107	fa[.]	Floating Add	fadd[.]	
108	fd[.]	Floating Divide	fdiv[.]	
108	fm[.]	Floating Multiply	fmul[.]	
109	fma[.]	Floating Multiply-Add	fmadd[.]	
109	fms[.]	Floating Multiply-Subtract	fmsub[.]	
110	fnma[.]	Floating Negative Multiply-Add	fnmadd[.]	
110	fnms[.]	Floating Negative Multiply-Subtract	fnmsub[.]	
107	fs[.]	Floating Subtract	fsub[.]	
Bk II	ics	Instruction Cache Synchronize	isync	Instruction Synchronize
33	l	Load	lwz	Load Word and Zero
40	lbrx	Load Byte-Reverse Indexed	lwbrx	Load Word Byte-Reverse Indexed
42	lm	Load Multiple	lmw	Load Multiple Word
44	lsi	Load String Immediate	lswi	Load String Word Immediate
44	lsx	Load String Indexed	lswx	Load String Word Indexed
33	lu	Load with Update	lwzu	Load Word and Zero with Update
33	lux	Load with Update Indexed	lwzux	Load Word and Zero with Update Indexed

Page / Bk	Power		PowerPC	
	Mnemonic	Instruction	Mnemonic	Instruction
33	lx	Load Indexed	lwzx	Load Word and Zero Indexed
Bk III	mtsri	Move To Segment Register Indirect	mtsrin	
55	muli	Multiply Immediate	mulli	Multiply Low Immediate
55	muls[o][.]	Multiply Short	mullw[o][.]	Multiply Low Word
64	oril	OR Immediate Lower	ori	OR Immediate
64	oriu	OR Immediate Upper	oris	OR Immediate Shifted
74	rlimi[.]	Rotate Left Immediate Then Mask Insert	rlwimi[.]	Rotate Left Word Immediate then Mask Insert
71	rlinm[.]	Rotate Left Immediate Then AND With Mask	rlwinm[.]	Rotate Left Word Immediate then AND with Mask
73	rlnm[.]	Rotate Left Then AND With Mask	rlwnm[.]	Rotate Left Word then AND with Mask
52	sf[o][.]	Subtract From	subfc[o][.]	Subtract From Carrying
53	sfe[o][.]	Subtract From Extended	subfe[o][.]	
52	sfi	Subtract From Immediate	subfic	Subtract From Immediate Carrying
53	sfme[o][.]	Subtract From Minus One Extended	subfme[o][.]	
54	sfze[o][.]	Subtract From Zero Extended	subfze[o][.]	
75	sl[.]	Shift Left	slw[.]	Shift Left Word
76	sr[.]	Shift Right	srw[.]	Shift Right Word
78	sra[.]	Shift Right Algebraic	sraw[.]	Shift Right Algebraic Word
77	srai[.]	Shift Right Algebraic Immediate	srawi[.]	Shift Right Algebraic Word Immediate
38	st	Store	stw	Store Word
41	stbrx	Store Byte-Reverse Indexed	stwbrx	Store Word Byte-Reverse Indexed
42	stm	Store Multiple	stmw	Store Multiple Word
45	stsi	Store String Immediate	stswi	Store String Word Immediate
45	stsx	Store String Indexed	stswx	Store String Word Indexed
38	stu	Store with Update	stwu	Store Word with Update
38	stux	Store with Update Indexed	stwux	Store Word with Update Indexed
38	stx	Store Indexed	stwx	Store Word Indexed
22	svca	Supervisor Call	sc	System Call
62	t	Trap	tw	Trap Word
61	ti	Trap Immediate	twi	Trap Word Immediate
Bk III	tlbi	TLB Invalidate Entry	tlbie	TLB Entry Invalidate
64	xoril	XOR Immediate Lower	xori	XOR Immediate
64	xoriu	XOR Immediate Upper	xoris	XOR Immediate Shifted

Appendix G. Incompatibilities with the Power Architecture

This section identifies the known incompatibilities that must be managed in the migration from the Power Architecture to the PowerPC Architecture. Some of the incompatibilities can, at least in principle, be detected by the processor, which could trap and let software simulate the Power operation. Others cannot be detected by the processor even in principle.

In general, the incompatibilities identified here are those that affect a Power application program: incompatiblities for instructions that can be used only by Power system programs are not necessarily discussed.

G.1 New Instructions, Formerly Privileged Instructions

Instructions new to PowerPC typically use opcode values (including extended opcode) that are illegal in Power. A few instructions that are privileged in Power (e.g., ***dclz***, called ***dcbz*** in PowerPC) have been made non-privileged in PowerPC. Any Power program that executes one of these now-valid or now-non-privileged instructions, expecting to cause the system illegal instruction error handler or the system privileged instruction error handler to be invoked, will not execute correctly on PowerPC.

G.2 Newly Privileged Instructions

The following instructions are non-privileged in Power but privileged in PowerPC.

mfmsr
mfsr

G.3 Reserved Bits in Instructions

These are shown with '/'s in the instruction layouts. In Power such bits are ignored by the processor. In PowerPC they must be 0 or the instruction form is invalid.

In several cases the PowerPC Architecture assumes that such bits in Power instructions are indeed 0. The cases include the following.

- ***cmpi, cmp, cmpli***, and ***cmpl*** assume that bit 10 in the Power instructions is 0.
- ***mtspr*** and ***mfspr*** assume that bits 16:20 in the Power instructions are 0.

G.4 Reserved Bits in Registers

Power defines these bits to be 0 on read, and either 0 or 1 on write. In PowerPC it is implementation dependent, for each bit, whether the bit is:

- 0 on read and ignored on write; or
- copied from source to target on both read and write.

G.5 Alignment Check

The Power MSR AL bit (bit 24) is no longer supported: the bit is reserved in PowerPC. The low-order bits of the EA are always used. (Notice that the value 0 — the normal value for a reserved SPR bit — means "ignore the low-order EA bits" in Power, and the value 1 means "use the low-order EA bits.") However, MSR bit 24 will not be assigned new meaning in the near future (see Book III, *PowerPC Operating Environment Architecture*), and software is permitted to write the value 1 to the bit.

G.6 Condition Register

The following instructions specify a field in the CR explicitly (via the BF field) and also have the Record bit. In PowerPC, if Rc=1 for these instructions the instruction form is invalid. In Power, if Rc=1 the instructions execute normally except as follows.

cmp	CR0 is undefined if Rc=1 and BF≠0
cmpl	CR0 is undefined if Rc=1 and BF≠0
mcrxr	CR0 is undefined if Rc=1 and BF≠0
fcmpu	CR1 is undefined if Rc=1
fcmpo	CR1 is undefined if Rc=1
mcrfs	CR1 is undefined if Rc=1 and BF≠1

G.7 Inappropriate use of LK and Rc bits

For the instructions listed below, if LK=1 or Rc=1 Power executes the instruction normally with the exception of setting the Link Register (if LK=1) or Condition Register Field 0 or 1 (if Rc=1) to an undefined value. In PowerPC such instruction forms are invalid.

PowerPC instruction form invalid if LK=1:

sc (***svc*** in Power)
the *Condition Register Logical* instructions
mcrf
isync (***ics*** in Power)

PowerPC instruction form invalid if Rc=1:

fixed-point X-form *Load* and *Store* instructions
fixed-point X-form *Compare* instructions
the X-form *Trap* instruction
mtspr, mfspr, mtcrf, mcrxr, mfcr
floating-point X-form *Load* and *Store* instructions
floating-point *Compare* instructions
mcrfs
dcbz (***dclz*** in Power)

G.8 BO Field

Power shows certain bits in the BO field — used by *Branch Conditional* instructions — as "x." Although the Power Architecture does not say how these bits are to be interpreted, they are in fact ignored by the processor. PowerPC treats these bits differently, as follows.

$BO_{0:3}$ PowerPC shows the bit as "z." (For the "branch always" encoding of the BO field, BO_4 is also shown as "z.") If a "z" bit is not zero the instruction form is invalid.

BO_4 This bit — which is shown as "x" in Power independent of the other four bits — is shown in PowerPC as "y" (except for the "branch always" encoding of the BO field). The "y" bit gives a hint about whether the branch is likely to be taken. If a Power program has the "wrong" value for this bit, the program will run correctly but performance may suffer.

G.9 Branch Conditional to Count Register

For the case in which the Count Register is decremented and tested (i.e., the case in which $BO_2=0$), Power specifies only that the branch target address is undefined, with the implication that the Count Register, and the Link Register if LK=1, are updated in the normal way. PowerPC considers this instruction form invalid.

G.10 System Call

There are several respects in which PowerPC is incompatible with Power for *System Call* instructions — which in Power are called *Supervisor Call* instructions.

- Power provides a version of the *Supervisor Call* instruction (bit 30 = 0) that allows instruction fetching to continue at any one of 128 locations. It is used for "fast SVCs". PowerPC provides no such version: if bit 30 of the instruction is 0 the instruction is reserved.
- Power provides a version of the *Supervisor Call* instruction (bits 30:31 = 0b11) that resumes instruction fetching at one location and sets the Link Register to the address of the next instruction. PowerPC provides no such version: if bit 31 of the instruction is 1 the instruction form is invalid.
- For Power, information from the MSR is saved in the Count Register. For PowerPC this information is saved in SRR 1.
- Power permits bits 16:29 of the instruction to be non-zero, while in PowerPC such an instruction form is invalid.

Architecture and Engineering Note

Bits 16:29 should be regarded as reserved for Power. As long as Power compatibility is required for this instruction, bits 16:29 should be ignored by the processor.

- Power saves the low-order 16 bits of the instruction, in the Count Register. PowerPC does not save them.
- The settings of MSR bits by the associated interrupt differ between Power and PowerPC: see *POWER Processor Architecture* and Book III, *PowerPC Operating Environment Architecture*.

G.11 Fixed-Point Exception Register (XER)

Bits 16:23 of the XER are reserved in PowerPC, while in Power they are defined and contain the comparison byte for the ***lscbx*** instruction (which PowerPC lacks).

Engineering Note

For reasons of compatibility with the Power Architecture, early implementations *must* handle XER bits 16:23 according to the second of the two permitted treatments of reserved bits in status and control registers. That is, early implementations must set the bits from the source value on write, and return the value last set for them on read.

G.12 Update Forms of Storage Access

PowerPC requires that RA not be equal to either RT (fixed-point *Load* only) or 0. If the restriction is violated the instruction form is invalid. Power permits these cases, and simply avoids saving the EA.

G.13 Multiple Register Loads

PowerPC requires that RA, and RB if present in the instruction format, not be in the range of registers to be loaded, while Power permits this and does not alter RA or RB in this case. (The PowerPC restriction applies even if RA=0, although there is no obvious benefit to the restriction in this case since RA is not used to compute the effective address if RA=0.) If the PowerPC restriction is violated, the instruction form is invalid. The instructions affected are:

lmw (***lm*** in Power)
lswi (***lsi*** in Power)
lswx (***lsx*** in Power)

Thus, for example, an ***lmw*** instruction that loads all 32 registers is valid in Power but is an invalid form in PowerPC.

G.14 Alignment for Load/Store Multiple

PowerPC requires the EA to be word-aligned, and yields an Alignment interrupt or boundedly undefined results if it is not. Power specifies that an Alignment interrupt occurs (if AL=1).

Engineering Note

If attempt is made to execute an ***lmw*** or ***stmw*** instruction having an incorrectly aligned effective address, early implementations *must* either correctly transfer the addressed bytes or cause an Alignment interrupt, for reasons of compatibility with the Power Architecture.

G.15 Load String Instructions

In PowerPC an ***lswx*** instruction with zero length leaves the content of RT undefined, while in Power the corresponding instruction (***lsx***) does not alter RT.

G.16 Synchronization

The ***sync*** instruction (called ***dcs*** in Power) and the ***isync*** instruction (called ***ics*** in Power) cause much more pervasive synchronization in PowerPC than in Power.

G.17 Move To/From SPR

There are several respects in which PowerPC is incompatible with Power for *Move To/From Special Purpose Register* instructions.

- The SPR field is ten bits long in PowerPC, but only five in Power (see also Section G.3, "Reserved Bits in Instructions" on page 165).
- ***mfspr*** can be used to read the Decrementer in problem state in Power, but only in privileged state in PowerPC.
- If the SPR value specified in the instruction is not one of the defined values, PowerPC considers the instruction form invalid. (In problem state, the allowed SPR values exclude those accessible only in privileged state.) Power does not alter any architected registers in this case, and generates a Privileged Instruction type Program interrupt if the instruction is executed in problem state and $SPR_0=1$.

Engineering Note

For reasons of compatibility with the Power Architecture, early implementations *must* cause an Illegal Instruction type Program interrupt for an attempt to execute an ***mtspr*** or ***mfspr*** instruction with $spr_{0:4}=0$ (which denotes the Power MQ register).

Similarly, early implementations *must* cause an Illegal Instruction type Program interrupt for an attempt to execute an ***mfspr*** instruction with $spr_{0:4}=4$ (which denotes reading the Real-Time Clock Upper in Power), $spr_{0:4}=5$ (which denotes reading the Real-Time Clock Lower in Power), or $spr_{0:4}=6$ (which denotes reading the Decrementer in Power).

G.18 Effects of Exceptions on FPSCR Bits FR and FI

For the following cases, Power does not say how FR and FI are set, while PowerPC preserves them for Invalid Operation Exceptions caused by *Compare* instructions and clears them otherwise.

Invalid Operation Exception (enabled or disabled)

Zero Divide Exception (enabled or disabled)

Disabled Overflow Exception

G.19 Floating-Point Store Instructions

Power uses $FPSCR_{UE}$ to help determine whether denormalization should be done, while PowerPC does not. Using $FPSCR_{UE}$ is in fact incorrect: if $FPSCR_{UE}=1$ and a denormalized single-precision number is copied from one storage location to another by means of ***lfs*** followed by ***stfs***, the two "copies" may not be the same.

G.20 Move From FPSCR

Power defines the high-order 32 bits of the result of ***mffs*** to be 0xFFFF_FFFF, while PowerPC says they are undefined.

G.21 Zeroing Bytes in the Data Cache

The ***dclz*** instruction of Power and the ***dcbz*** instruction of PowerPC have the same opcode. However, the functions differ in the following respects.

- ***dclz*** clears a line while ***dcbz*** clears a block.
- ***dclz*** saves the EA in RA (if RA≠0) while ***dcbz*** does not.
- ***dclz*** is privileged while ***dcbz*** is not.

G.22 Floating-Point Load/Store to Direct-Store Segment

In Power a floating-point *Load* or *Store* instruction to a direct-store segment causes a Data Storage interrrupt, while in PowerPC the instruction either executes correctly or causes an Alignment interrupt.

G.23 Segment Register Instructions

The definitions of the four Segment Register instructions (***mtsr***, ***mtsrin***, ***mfsr***, and ***mfsrin***) differ in two respects between Power and PowerPC. Instructions similar to ***mtsrin*** and ***mfsrin*** are called ***mtsri*** and ***mfsri*** in Power.

privilege: ***mfsr*** and ***mfsri*** are problem state instructions in Power, while ***mfsr*** and ***mfsrin*** are privileged in PowerPC.

function: the "indirect" instructions (***mtsri*** and ***mfsri***) in Power use an RA register in computing the Segment Register number, and the computed EA is stored into RA (if RA≠0 and RA≠RT), while in PowerPC ***mtsrin*** and ***mfsrin*** have no RA field and EA is not stored.

mtsr, ***mtsrin*** (***mtsri***), and ***mfsr*** have the same opcodes in PowerPC as in Power. ***mfsri*** (Power) and ***mfsrin*** (PowerPC) have different opcodes.

G.24 TLB Entry Invalidation

The ***tlbi*** instruction of Power and the ***tlbie*** instruction of PowerPC have the same opcode. However, the functions differ in the following respects.

- ***tlbi*** computes the EA as (RA|0) + (RB), while ***tlbie*** lacks an RA field and computes the EA as (RB).
- ***tlbi*** saves the EA in RA (if RA≠0), while ***tlbie*** lacks an RA field and does not save the EA.

G.25 Floating-Point Interrupts

Both architectures use MSR bit 20 to control the generation of interrupts for floating-point enabled exceptions. However, in PowerPC this bit is part of a two-bit value which controls the occurrence, precision, and recoverability of the interrupt, while in Power this bit is used independently to control the occurence of the interrupt (in Power all floating-point interrupts are precise).

G.26 Timing Facilities

G.26.1 Real-Time Clock

The Power Real-Time Clock is not supported in PowerPC. Instead, PowerPC provides a Time Base. Both the RTC and the TB are 64-bit Special Purpose Registers, but they differ in the following respects.

- The RTC counts seconds and nanoseconds, while the TB counts "ticks." The ticking rate of the RTC is implementation-dependent.
- The RTC increments discontinuously: 1 is added to RTCU when the value in RTCL passes 999_999_999. The TB increments continuously: 1 is added to TBU when the value in TBL passes 0xFFFF_FFFF.
- The RTC is written and read by the ***mtspr*** and ***mfspr*** instructions, using SPR numbers that denote the RTCU and RTCL. The TB is written by the ***mtspr*** instruction (using new SPR numbers), and read by the new ***mftb*** instruction.
- The SPR numbers that denote Power's RTCL and RTCU are invalid in PowerPC.
- The RTC is guaranteed to increment at least once in the time required to execute ten *Add Immediate* instructions. No analogous guarantee is made for the TB.
- Not all bits of RTCL need be implemented, while all bits of the TB must be implemented.

G.26.2 Decrementer

The PowerPC Decrementer differs from the Power Decrementer in the following respects.

- The PowerPC DEC decrements at the same rate that the TB increments, while the Power Decrementer decrements every nanosecond (which is the same rate that the RTC increments).
- Not all bits of the Power DEC need be implemented, while all bits of the PowerPC DEC must be implemented.
- The interrupt caused by the DEC has its own interrupt vector location in PowerPC, but is considered an External interrupt in Power.

G.27 Deleted Instructions

The following instructions are part of the Power Architecture but have been dropped from the PowerPC Architecture.

abs	Absolute
clcs	Cache Line Compute Size
clf	Cache Line Flush
cli	Cache Line Invalidate
dclst	Data Cache Line Store
div	Divide
divs	Divide Short
doz	Difference Or Zero
dozi	Difference Or Zero Immediate
lscbx	Load String And Compare Byte Indexed
maskg	Mask Generate
maskir	Mask Insert From Register
mfsri	Move From Segment Register Indirect
mul	Multiply
nabs	Negative Absolute
rac	Real Address Compute
rlmi	Rotate Left Then Mask Insert
rrib	Rotate Right And Insert Bit
sle	Shift Left Extended
sleq	Shift Left Extended With MQ
sliq	Shift Left Immediate With MQ
slliq	Shift Left Long Immediate With MQ
sllq	Shift Left Long With MQ
slq	Shift Left With MQ
sraiq	Shift Right Algebraic Immediate With MQ
sraq	Shift Right Algebraic With MQ
sre	Shift Right Extended
srea	Shift Right Extended Algebraic
sreq	Shift Right Extended With MQ
sriq	Shift Right Immediate With MQ
srliq	Shift Right Long Immediate With MQ
srlq	Shift Right Long With MQ
srq	Shift Right With MQ
svc[l]	Supervisor Call, with SA=0

Note: Many of these instructions use the MQ register. The MQ is not defined in the PowerPC Architecture.

G.28 Discontinued Opcodes

The opcodes listed below are defined in the Power Architecture but have been dropped from the PowerPC Architecture. The list contains the old mnemonic (MNEM), the primary opcode (PRI), and the extended opcode (XOP) if appropriate.

MNEM	PRI	XOP
abs	31	360
clcs	31	531
clf	31	118
cli	31	502
dclst	31	630
div	31	331
divs	31	363
doz	31	264
dozi	09	–
lscbx	31	277
maskg	31	29
maskir	31	541
mfsri	31	627
mul	31	107
nabs	31	488
rac	31	818
rlmi	22	–
rrib	31	537
sle	31	153
sleq	31	217
sliq	31	184
slliq	31	248
sllq	31	216
slq	31	152
sraiq	31	952
sraq	31	920
sre	31	665
srea	31	921
sreq	31	729
sriq	31	696
srliq	31	760
srlq	31	728
srq	31	664
svc[l]	17	0

Assembler Note

It might be helpful to current software writers for the Assembler to flag the discontinued Power instructions.

G.29 Rios-2 Compatibility

Editors' Note

Rios-2 is an unannounced IBM product. If this Book is published before the Rios-2 product is announced, this section should be omitted.

The Rios-2 instruction set is a superset of the Power instruction set. Some of the instructions added for Rios-2 are included in the PowerPC Architecture. Those that have been renamed in the PowerPC Architecture are listed in this section, as are the new Rios-2 instructions that are not included in the PowerPC Architecture.

Other incompatibilities are also listed.

G.29.1 Cross-Reference for Changed Rios-2 Mnemonics

The following table lists the new Rios-2 instruction mnemonics that have been changed in the PowerPC User Instruction Set Architecture, sorted by Rios-2 mnemonic.

To determine the PowerPC mnemonic for one of these Rios-2 mnemonics, find the Rios-2 mnemonic in the second column of the table: the remainder of the line gives the PowerPC mnemonic and the page on which the instruction is described, as well as the instruction names.

Rios-2 mnemonics that have not changed are not listed.

Page	Rios-2		PowerPC	
	Mnemonic	Instruction	Mnemonic	Instruction
113	fcir[.]	Floating Convert Double to Integer with Round	fctiw[.]	Floating Convert to Integer Word
113	fcirz[.]	Floating Convert Double to Integer with Round to Zero	fctiwz[.]	Floating Convert to Integer Word with round toward Zero

G.29.2 Floating-Point Conversion to Integer

The ***fcir*** and ***fcirz*** instructions of Rios-2 have the same opcodes as do the ***fctiw*** and ***fctiwz*** instructions, respectively, of PowerPC. However, the functions differ in the following respects.

- ***fcir*** and ***fcirz*** set the high-order 32 bits of the target FPR to zero, while ***fctiw*** and ***fctiwz*** set them to an undefined value.
- Except for enabled Invalid Operation Exceptions, ***fcir*** and ***fcirz*** set the FPRF field of the FPSCR based on the result, while ***fctiw*** and ***fctiwz*** set it to an undefined value.
- ***fcir*** and ***fcirz*** do not affect the VXSNAN bit of the FPSCR, while ***fctiw*** and ***fctiwz*** do.

G.29.3 Storage Ordering

Rios-2 uses MSR bit 28 to control storage ordering. This bit is reserved in PowerPC, and no corresponding control is provided.

G.29.4 Floating-Point Interrupts

Both architectures use MSR bits 20 and 23 to control the generation of interrupts for floating-point enabled exceptions. However, in PowerPC these bits comprise a two-bit value which controls the occurrence, precision, and recoverability of the interrupt, while in Rios-2 these bits are used independently to control the occurence (bit 20) and the precision (bit 23) of the interrupt. Moreover, in PowerPC all floating-point interrupts are considered Program interrupts, while in Rios-2 imprecise floating-point interrupts have their own interrupt vector location.

G.29.5 Trace Interrupts

The interrupt vector location differs between the two architectures.

G.29.6 Deleted Instructions

The following instructions are new in the Rios-2 Architecture but have been dropped from the PowerPC Architecture.

lfq	Load Floating-Point Quad
lfqu	Load Floating-Point Quad with Update
lfqux	Load Floating-Point Quad with Update Indexed
lfqx	Load Floating-Point Quad Indexed
stfq	Store Floating-Point Quad
stfqu	Store Floating-Point Quad with Update
stfqux	Store Floating-Point Quad with Update Indexed
stfqx	Store Floating-Point Quad Indexed

G.29.7 Discontinued Opcodes

The opcodes listed below are new in the Rios-2 Architecture but have been dropped from the PowerPC Architecture. The list contains the old mnemonic (MNEM), the primary opcode (PRI), and the extended opcode (XOP) if appropriate.

MNEM	PRI	XOP
lfq	56	–
lfqu	57	–
lfqux	31	823
lfqx	31	791
stfq	60	–
stfqu	61	–
stfqux	31	951
stfqx	31	919

Appendix H. New Instructions

The following instructions in the PowerPC Architecture are new: they are not in the Power Architecture.

They are listed in three groups, according to whether they exist in all PowerPC implementations, only in 64-bit implementations, or only in 32-bit implementations.

The following instructions are optional: ***eciwx, ecowx, fres, frsqrte, fsel, fsqrt***[s], ***slbia, slbie, slbiex, stfiwx, tlbia, tlbiex, tlbsync.***

H.1 New Instructions for All Implementations

dcbf	Data Cache Block Flush
dcbi	Data Cache Block Invalidate
dcbst	Data Cache Block Store
dcbt	Data Cache Block Touch
dcbtst	Data Cache Block Touch for Store
divw	Divide Word
divwu	Divide Word Unsigned
eciwx	External Control In Word Indexed
ecowx	External Control Out Word Indexed
eieio	Enforce In-order Execution of I/O
extsb	Extend Sign Byte
fadds	Floating Add Single
fctiw	Floating Convert To Integer Word
fctiwz	Floating Convert To Integer Word with round toward Zero
fdivs	Floating Divide Single
fmadds	Floating Multiply-Add Single
fmsubs	Floating Multiply-Subtract Single
fmuls	Floating Multiply Single
fnmadds	Floating Negative Multiply-Add Single
fnmsubs	Floating Negative Multiply-Subtract Single
fres	Floating Reciprocal Estimate Single
frsqrte	Floating Reciprocal Square Root Estimate
fsel	Floating Select
fsqrt[s]	Floating Square Root [Single]
fsubs	Floating Subtract Single
icbi	Instruction Cache Block Invalidate
lwarx	Load Word And Reserve Indexed
mftb	Move From Time Base
mulhw	Multiply High Word
mulhwu	Multiply High Word Unsigned
stfiwx	Store Floating-Point as Integer Word Indexed
stwcx.	Store Word Conditional Indexed
subf	Subtract From
tlbia	TLB Invalidate All
tlbiex	TLB Invalidate Entry by Index
tlbsync	TLB Synchronize

H.2 New Instructions for 64-Bit Implementations Only

cntlzd	Count Leading Zeros Doubleword
divd	Divide Doubleword
divdu	Divide Doubleword Unsigned
extsw	Extend Sign Word
fcfid	Floating Convert From Integer Doubleword
fctid	Floating Convert To Integer Doubleword
fctidz	Floating Convert To Integer Doubleword with round toward Zero
lwa	Load Word Algebraic
lwaux	Load Word Algebraic with Update Indexed
lwax	Load Word Algebraic Indexed
ld	Load Doubleword
ldarx	Load Doubleword And Reserve Indexed
ldu	Load Doubleword with Update
ldux	Load Doubleword with Update Indexed
ldx	Load Doubleword Indexed
mulhd	Multiply High Doubleword
mulhdu	Multiply High Doubleword Unsigned
mulld	Multiply Low Doubleword
rldcl	Rotate Left Doubleword then Clear Left
rldcr	Rotate Left Doubleword then Clear Right
rldic	Rotate Left Doubleword Immediate then Clear
rldicl	Rotate Left Doubleword Immediate then Clear Left
rldicr	Rotate Left Doubleword Immediate then Clear Right
rldimi	Rotate Left Doubleword Immediate then Mask Insert
slbia	SLB Invalidate All
slbie	SLB Invalidate Entry
slbiex	SLB Invalidate Entry by Index
sld	Shift Left Doubleword
srad	Shift Right Algebraic Doubleword
sradi	Shift Right Algebraic Doubleword Immediate
srd	Shift Right Doubleword
std	Store Doubleword
stdcx.	Store Doubleword Conditional Indexed
stdu	Store Doubleword with Update
stdux	Store Doubleword with Update Indexed
stdx	Store Doubleword Indexed
td	Trap Doubleword
tdi	Trap Doubleword Immediate

H.3 New Instructions for 32-Bit Implementations Only

mfsrin Move From Segment Register Indirect

H.4 Instructions with Different Semantics

The following instructions, which are all privileged, have the same opcode in both PowerPC and Power, but perform differently.

PowerPC	Power
dcbz	***dclz***
tlbie	***tlbi***

Appendix I. Illegal Instructions

With the exception of the instruction consisting entirely of binary 0's, the instructions in this class are available for future extensions of the PowerPC Architecture: that is, some future version of the PowerPC Architecture may define any of these instructions to perform new functions.

The following primary opcodes are illegal.

1, 4, 5, 6, 56, 57, 60, 61

In addition, the following primary opcodes are illegal for 32-bit implementations (they are defined only for 64-bit implementations).

2, 30, 58, 62

The following primary opcodes have unused extended opcodes. Their unused extended opcodes can be determined from the opcode maps in Appendix K, "Opcode Maps" on page 179. Extended opcodes for instructions that are defined only for 64-bit implementations are illegal in 32-bit implementations, and extended opcodes for instructions that are defined only for 32-bit implementations are illegal in 64-bit implementations. All unused extended opcodes are illegal.

17, 19, 30[1], 31, 59, 62[1], 63

[1] Applies only for 64-bit implementations (illegal primary opcode for 32-bit implementations)

An instruction consisting entirely of binary 0's is illegal, and is guaranteed to be illegal in all future versions of this architecture.

Appendix J. Reserved Instructions

The instructions in this class are allocated to specific purposes that are outside the scope of the PowerPC User Instruction Set Architecture, PowerPC Virtual Environment Architecture, and PowerPC Operating Environment Architecture.

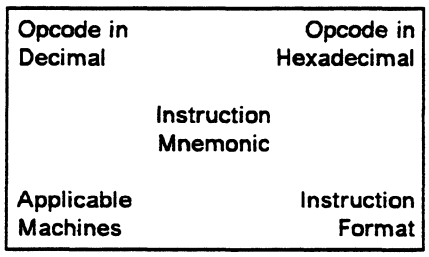
The following types of instruction are included in this class.

1. Instructions for the Power Architecture which have not been included in the PowerPC Architecture. These are listed in Appendix G, "Incompatibilities with the Power Architecture" on page 165.
2. Implementation-specific instructions used to conform to the PowerPC Architecture specifications.
3. The instruction with primary opcode 0, when the instruction does not consist entirely of binary 0's.
4. Any other instructions contained in Book IV, *PowerPC Implementation Features* for any implementation, which are not defined in the PowerPC User Instruction Set Architecture, PowerPC Virtual Environment Architecture, nor PowerPC Operating Environment Architecture.

Appendix K. Opcode Maps

This section contains tables showing the opcodes and extended opcodes in all members of the Power architecture family.

For the primary opcode table (Table 11 on page 181), each cell is in the following format.



"Applicable Machines" identifies the Power architecture family members that recognize the opcode, encoded as follows:

P	PowerPC
2	Rios-2
O	Original Power (RS/6000)
All	All of the above

Editors' Note

Rios-2 is an unannounced IBM product. If this Book is published before the Rios-2 product is announced, the code "2" should be omitted from the tables in this appendix, as should all instructions that exist only in Rios-2.

The extended opcode tables show the extended opcode in decimal, the instruction mnemonic, the applicable machines, and the instruction format. These tables appear in order of primary opcode within two groups. The first group consists of the primary opcodes that have small extended opcode fields (2-4 bits), namely 30, 56, 57, 58, 60, 61, and 62. The second group consists of primary opcodes that have 10-bit extended opcode fields. The tables for the second group are rotated.

In the extended opcode tables several special markings are used.

- A prime (') following an instruction mnemonic denotes an additional cell, after the lowest-numbered one, used by the instruction. For example, ***subfc*** occupies cells 8 and 520 of primary opcode 31, with the former corresponding to OE=0 and the latter to OE=1. Similarly, ***sradi*** occupies cells 826 and 827, with the former corresponding to $sh_5=0$ and the latter to $sh_5=1$ (the 9-bit extended opcode 413, shown on page 77, excludes the sh_5 bit).
- Two vertical bars (||) are used instead of primed mnemonics when an instruction occupies an entire column of a table. The instruction mnemonic is repeated in the last cell of the column.
- For primary opcode 31, an asterisk (*) in a cell that would otherwise be empty means that the cell is reserved because it is "overlaid," by a fixed-point or *Storage Access* instruction having only a primary opcode, by an instruction having an extended opcode in primary opcode 30, 58, or 62, or by a potential instruction in any of the categories just mentioned. The overlaying instruction, if any, is also shown. A cell thus reserved should not be assigned to an instruction having primary opcode 31. (The overlaying is a consequence of opcode decoding for fixed-point instructions: the primary opcode, and the extended opcode if any, are mapped internally to a 10-bit "compressed opcode" for ease of subsequent decoding.)

An empty cell, or a cell containing only an asterisk, corresponds to an illegal instruction.

When instruction names and/or mnemonics differ among the family members, the PowerPC terminology is used.

The instruction consisting of 32 0-bits causes the system illegal instruction error handler to be invoked for all members of the Power family, and this is likely to remain true in future models (it is guaranteed in the PowerPC architecture). An instruction with primary opcode 0 but not consisting entirely of 0-bits is reserved.

Table 11. Primary Opcodes				
00 00 Illegal, Reserved All	01 01	02 02 tdi P D	03 03 twi All D	 Trap Doubleword Immediate Trap Word Immediate
04 04	05 05	06 06	07 07 mulli All D	 Multiply Low Immediate
08 08 subfic All D	09 09 dozi 2O D	10 0A cmpli All D	11 0B cmpi All D	Subtract From Immediate Carrying Difference or Zero Immediate Compare Logical Immediate Compare Immediate
12 0C addic All D	13 0D addic. All D	14 0E addi All D	15 0F addis All D	Add Immediate Carrying Add Immediate Carrying and Record Add Immediate Add Immediate Shifted
16 10 bc All B	17 11 sc All SC	18 12 b All I	19 13 CR ops, etc. All XL	Branch Conditional System Call Branch See Table 19 on page 184
20 14 rlwimi All M	21 15 rlwinm All M	22 16 rlmi 2O M	23 17 rlwnm All M	Rotate Left Word Imm. then Mask Insert Rotate Left Word Imm. then AND with Mask Rotate Left then Mask Insert Rotate Left Word then AND with Mask
24 18 ori All D	25 19 oris All D	26 1A xori All D	27 1B xoris All D	OR Immediate OR Immediate Shifted XOR Immediate XOR Immediate Shifted
28 1C andi. All D	29 1D andis. All D	30 1E FX Dwd Rot P MD[S]	31 1F FX Extended Ops All	AND Immediate AND Immediate Shifted See Table 12 on page 182 See Table 20 on page 186
32 20 lwz All D	33 21 lwzu All D	34 22 lbz All D	35 23 lbzu All D	Load Word and Zero Load Word and Zero with Update Load Byte and Zero Load Byte and Zero with Update
36 24 stw All D	37 25 stwu All D	38 26 stb All D	39 27 stbu All D	Store Word Store Word with Update Store Byte Store Byte with Update
40 28 lhz All D	41 29 lhzu All D	42 2A lha All D	43 2B lhau All D	Load Half and Zero Load Half and Zero with Update Load Half Algebraic Load Half Algebraic with Update
44 2C sth All D	45 2D sthu All D	46 2E lmw All D	47 2F stmw All D	Store Half Store Half with Update Load Multiple Word Store Multiple Word
48 30 lfs All D	49 31 lfsu All D	50 32 lfd All D	51 33 lfdu All D	Load Floating-Point Single Load Floating-Point Single with Update Load Floating-Point Double Load Floating-Point Double with Update
52 34 stfs All D	53 35 stfsu All D	54 36 stfd All D	55 37 stfdu All D	Store Floating-Point Single Store Floating-Point Single with Update Store Floating-Point Double Store Floating-Point Double with Update
56 38 lfq, 3 illegal 2 DS	57 39 lfqu, 3 illegal 2 DS	58 3A FX DS-form Loads P DS	59 3B FP Single Extended Ops P A	See Table 13 on page 183 See Table 14 on page 183 See Table 15 on page 183 See Table 21 on page 188
60 3C stfq, 3 illegal 2 DS	61 3D stfqu, 3 illegal 2 DS	62 3E FX DS-Form Stores P DS	63 3F FP Double Extended Ops All	See Table 16 on page 183 See Table 17 on page 183 See Table 18 on page 183 See Table 22 on page 190

Table 12. Extended Opcodes for Primary Opcode 30 (instruction bits 27:30)

	00	01	10	11
00	0 *rldicl* P MD	1 *rldicl'* P MD	2 *rldicr* P MD	3 *rldicr'* P MD
01	4 *rldic* P MD	5 *rldic'* P MD	6 *rldimi* P MD	7 *rldimi'* P MD
10	8 *rldcl* P MDS	9 *rldcr* P MDS		
11				

Table 13. Extended Opcodes for Primary Opcode 56 (instruction bits 30:31)

	0	1
0	0 ***lfq*** 2 DS	
1		

Table 14. Extended Opcodes for Primary Opcode 57 (instruction bits 30:31)

	0	1
0	0 ***lfqu*** 2 DS	
1		

Table 15. Extended Opcodes for Primary Opcode 58 (instruction bits 30:31)

	0	1
0	0 ***ld*** P DS	1 ***ldu*** P DS
1	2 ***lwa*** P DS	

Table 16. Extended Opcodes for Primary Opcode 60 (instruction bits 30:31)

	0	1
0	0 ***stfq*** 2 DS	
1		

Table 17. Extended Opcodes for Primary Opcode 61 (instruction bits 30:31)

	0	1
0	0 ***stfqu*** 2 DS	
1		

Table 18. Extended Opcodes for Primary Opcode 62 (instruction bits 30:31)

	0	1
0	0 ***std*** P DS	1 ***stdu*** P DS
1		

Table 19 (Page 1 of 2). Extended Opcodes for Primary Opcode 19 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000	0 *mcrf* All XL																16 *bclr* All XL															
00001		33 *crnor* All XL																	50 *rfi* All XL													
00010																			82 *rfsvc* 2O XL													
00011																																
00100		129 *crandc* All XL																					150 *isync* All XL									
00101																																
00110		193 *crxor* All XL																														
00111		225 *crnand* All XL																														
01000		257 *crand* All XL																														
01001		289 *creqv* All XL																														
01010																																
01011																																
01100																																
01101		417 *crorc* All XL																														
01110		449 *cror* All XL																														
01111																																

Table 19 (Page 2 of 2). Extended Opcodes for Primary Opcode 19 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
10000																	528 *bcctr* All XL														
10001																																
10010																																
10011																																
10100																																
10101																																
10110																																
10111																																
11000																																
11001																																
11010																																
11011																																
11100																																
11101																																
11110																																
11111																																

Table 20 (Page 1 of 2). Extended Opcodes for Primary Opcode 31 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000	0 *cmp* All X				4 *tw* All X				8 *subfc* All XO	9 *mulhdu* P XO	10 *addc* All XO	11 *mulhwu* P XO				15 *Res0*[A] All				19 *mfcr* All X	20 *lwarx* P X	21 *ldx* P X		23 *lwzx* All X	24 *slw* All X		26 *cntlzw* All X	27 *sld* P X	28 *and* All X	29 *maskg* 2O X	30 *rldicl*[R] P MD	
00001	32 *cmpl* All X								40 *subf* P XO							47 [A]						53 *ldux* P X	54 *dcbst* P X	55 *lwzux* All X			58 *cntlzd* P X		60 *andc* All X		62 *rldicl*[R] P MD	
00010					68 *td* P X					73 *mulhd* P XO		75 *mulhw* P XO				79 *tdi*[A] P D				83 *mfmsr* All X	84 *ldarx* P X		86 *dcbf* P X	87 *lbzx* All X							94 *rldicr*[R] P MD	
00011									104 *neg* All XO			107 *mul* 2O XO				111 *twi*[A] All D							118 *clf* 2O X	119 *lbzux* All X					124 *nor* All X		126 *rldicr*[R] P MD	
00100									136 *subfe* All XO		138 *adde* All XO					143 [A]	144 *mtcrf* All XFX		146 *mtmsr* All X			149 *stdx* P X	150 *stwcx* P X	151 *stwx* All X	152 *slq* 2O X	153 *sle* 2O X					158 *rldic*[A] P MD	159 *riwinm*[A] All M
00101																175 [A]						181 *stdux* P X		183 *stwux* All X	184 *sliq* 2O X						190 *rldic*[A] P MD	191 *riwinm*[A] All M
00110									200 *subfze* All XO		202 *addze* All XO					207 [A]			210 *mtsr* All X				214 *stdcx.* P X	215 *stbx* All X	216 *sliq* 2O X	217 *sleq* 2O X					222 *rldimi*[A] P MD	223 *rlmi*[A] 2O M
00111									232 *subfme* All XO	233 *mulld* P XO	234 *addme* All XO	235 *mullw* All XO				239 *mulli*[A] All D			242 *mtsrin* All X				246 *dcbtst* P X	247 *stbux* All X	248 *slliq* 2O X						254 *rldimi*[A] P MD	255 *riwnm*[A] All M
01000									264 *doz* 2O XO		266 *add* All XO					271 *subfic*[A] All D						277 *lscbx* 2O X	278 *dcbt* P X	279 *lhzx* All X					284 *eqv* All X		286 *rldcl*[R] P MDS	287 *ori*[A] All D
01001																303 *dozi*[A] 2O D			306 *tlbie* All X				310 *eciwx* P X	311 *lhzux* All X					316 *xor* All X		318 *rldcl*[R] P MDS	319 *oris*[A] All D
01010												331 *div* 2O XO				335 *cmpli*[A] All D			338 *tlbiex* P X	339 *mfspr* All XFX		341 *lwax* P X		343 *lhax* All X							350 [A]	351 *xori*[A] All D
01011									360 *abs* 2O XO			363 *divs* 2O XO				367 *cmpi*[A] All D			370 *tlbia* P X	371 *mftb* P XFX		373 *lwaux* P X		375 *lhaux* All X							382 [A]	383 *xoris*[A] All D
01100																399 *addic*[R] All D								407 *sthx* All X					412 *orc* All X		414 [A]	415 *andi.*[A] All D
01101																431 *addic.*[A] All D			434 *slbie* P X				438 *ecowx* P X	439 *sthux* All X					444 *or* All X		446 [A]	447 *andis.*[A] All D
01110										457 *divdu* P XO		459 *divwu* P XO				463 *addi*[A] All D			466 *slbiex* P X	467 *mtspr* All XFX			470 *dcbi* P X	471 *lmw*[A] All D					476 *nand* All X		478 [A]	
01111									488 *nabs* 2O XO	489 *divd* P XO		491 *divw* P XO				495 *addis*[R] All D			498 *slbia* P X				502 *cli* 2O X	503 *stmw*[A] All D							510 [A]	

Table 20 (Page 2 of 2). Extended Opcodes for Primary Opcode 31 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
10000	512 *mcrxr* All X								520 *subfc'* All XO	521 *mulhdu'* P XO	522 *addc'* All XO	523 *mulhwu'* P XO								531 *clcs* 2O X		533 *lswx* All X	534 *lwbrx* All X	535 *lfsx* All X	536 *srw* All X	537 *rrib* 2O X		539 *srd* P X		541 *maskir* 2O X		
10001									552 *subf'* P XO														566 *tlbsync* P X	567 *lfsux* All X								
10010										585 *mulhd'* P XO		587 *mulhw'* P XO								595 *mfsr* All X		597 *lswi* All X	598 *sync* All X	599 *lfdx* All X								
10011									616 *neg'* All XO			619 *mul'* 2O XO								627 *mfsri* 2O X			630 *dclst* 2O X	631 *lfdux* All X								
10100									648 *subfe'* All XO		650 *adde'* All XO									659 *mfsrin* P X		661 *stswx* All X	662 *stwbrx* All X	663 *stfsx* All X	664 *srq* 2O X	665 *sre* 2O X						
10101																								695 *stfsux* All X	696 *sriq* 2O X							
10110									712 *subfze'* All XO		714 *addze'* All XO											725 *stswi* All X		727 *stfdx* All X	728 *srlq* 2O X	729 *sreq* 2O X						
10111									744 *subfme'* All XO	745 *mulld'* P XO	746 *addme'* All XO	747 *mullw'* All XO												759 *stfdux* All X	760 *srliq* 2O X							
11000									776 *doz'* 2O XO		778 *add'* All XO												790 *lhbrx* All X	791 *lfqx* 2 X	792 *sraw* All X		794 *srad* P X				
11001																			818 *rac* 2O X					823 *lfqux* 2 X	824 *srawi* All X		826 *sradi* P XS	827 *sradi'* P XS				
11010												843 *div'* 2O XO											854 *eieio* P X								
11011									872 *abs'* 2O XO			875 *divs'* 2O XO																				
11100																							918 *sthbrx* All X	919 *stfqx* 2 X	920 *sraq* 2O X	921 *srea* 2O X	922 *extsh* All X				
11101																								951 *stfqux* 2 X	952 *sraiq* 2O X		954 *extsb* P X				
11110										969 *divdu* P XO		971 *divwu* P XO											982 *icbi* P X	983 *stfiwx* P X			986 *extsw* P X				
11111									1000 *nabs'* 2O XO	1001 *divd'* P XO		1003 *divw'* P XO											1014 *dcbz* All X								

Table 21 (Page 1 of 2). Extended Opcodes for Primary Opcode 59 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000																			18 *fdivs* [P A]		20 *fsubs* [P A]	21 *fadds* [P A]	22 *fsqrts* [P A]		24 *fres* [P A]	25 *fmuls* [P A]			28 *fmsubs* [P A]	29 *fmadds* [P A]	30 *fnmsubs* [P A]	31 *fnmadds* [P A]
00001																			‖		‖	‖	‖		‖	‖			‖	‖	‖	‖
00010																			‖		‖	‖	‖		‖	‖			‖	‖	‖	‖
00011																			‖		‖	‖	‖		‖	‖			‖	‖	‖	‖
00100																			‖		‖	‖	‖		‖	‖			‖	‖	‖	‖
00101																			‖		‖	‖	‖		‖	‖			‖	‖	‖	‖
00110																			‖		‖	‖	‖		‖	‖			‖	‖	‖	‖
00111																			‖		‖	‖	‖		‖	‖			‖	‖	‖	‖
01000																			‖		‖	‖	‖		‖	‖			‖	‖	‖	‖
01001																			‖		‖	‖	‖		‖	‖			‖	‖	‖	‖
01010																			‖		‖	‖	‖		‖	‖			‖	‖	‖	‖
01011																			‖		‖	‖	‖		‖	‖			‖	‖	‖	‖
01100																			‖		‖	‖	‖		‖	‖			‖	‖	‖	‖
01101																			‖		‖	‖	‖		‖	‖			‖	‖	‖	‖
01110																			‖		‖	‖	‖		‖	‖			‖	‖	‖	‖
01111																			‖		‖	‖	‖		‖	‖			‖	‖	‖	‖

Table 21 (Page 2 of 2). Extended Opcodes for Primary Opcode 59 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
10000																																
10001																																
10010																																
10011																																
10100																																
10101																																
10110																																
10111																																
11000																																
11001																																
11010																																
11011																																
11100																																
11101																																
11110																																
11111																			*fdivs*		*fsubs*	*fadds*	*fsqrts*		*fres*	*fmuls*			*fmsubs*	*fmadds*	*fnmsubs*	*fnmadds*

Table 22 (Page 1 of 2). Extended Opcodes for Primary Opcode 63 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000	0 *fcmpu* All X												12 *frsp* All X		14 *fctiw* P2 X	15 *fctiwz* P2 X			18 *fdiv* All A		20 *fsub* All A	21 *fadd* All A	22 *fsqrt* P2 A	23 *fsel* P A		25 *fmul* All A	26 *frsqrte* P A		28 *fmsub* All A	29 *fmadd* All A	30 *fnmsub* All A	31 *fnmadd* All A
00001	32 *fcmpo* All X						38 *mtfsb1* All X		40 *fneg* All X																							
00010	64 *mcrfs* All X						70 *mtfsb0* All X		72 *fmr* All X																							
00011																																
00100							134 *mtfsfi* All X		136 *fnabs* All X																							
00101																																
00110																																
00111																																
01000									264 *fabs* All X																							
01001																																
01010																																
01011																																
01100																																
01101																																
01110																																
01111																																

Table 22 (Page 2 of 2). Extended Opcodes for Primary Opcode 63 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
10000																																
10001																																
10010								583 *mtfs* All X																								
10011																																
10100																																
10101																																
10110								711 *mtfsf* All XFL																								
10111																																
11000																																
11001															814 *fctid* P X	815 *fctidz* P X																
11010															846 *fctid* P X																	
11011																																
11100																																
11101																																
11110																																
11111																			*fdiv*		*fsub*	*fadd*	*fsqrt*	*fsel*		*fmul*	*frsqrte*		*fmsub*	*fmadd*	*fnmsub*	*fnmadd*

Appendix L. PowerPC Instruction Set Sorted by Opcode

This appendix lists all the instructions in the PowerPC Architecture. A page number is shown for instructions that are defined in this Book (Book I, *PowerPC User Instruction Set Architecture*), and the Book number is shown for instructions that are defined in other Books (Book II, *PowerPC Virtual Environment Architecture*, and Book III, *PowerPC Operating Environment Architecture*). If an instruction is defined in more than one Book, the lowest-numbered Book is used.

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
D	2		()	61	tdi	Trap Doubleword Immediate
D	3			61	twi	Trap Word Immediate
D	7			55	mulli	Multiply Low Immediate
D	8		SR	52	subfic	Subtract From Immediate Carrying
D	10			60	cmpli	Compare Logical Immediate
D	11			59	cmpi	Compare Immediate
D	12		SR	51	addic	Add Immediate Carrying
D	13		SR	51	addic.	Add Immediate Carrying and Record
D	14			50	addi	Add Immediate
D	15			50	addis	Add Immediate Shifted
B	16		CT	20	bc[l][a]	Branch Conditional
SC	17	1		22	sc	System Call
I	18			20	b[l][a]	Branch
XL	19	0		25	mcrf	Move Condition Register Field
XL	19	16	CT	21	bclr[l]	Branch Conditional to Link Register
XL	19	33		24	crnor	Condition Register NOR
XL	19	50		Bk III	rfi	Return From Interrupt
XL	19	129		24	crandc	Condition Register AND with Complement
XL	19	150		Bk II	isync	Instruction Synchronize
XL	19	193		23	crxor	Condition Register XOR
XL	19	225		23	crnand	Condition Register NAND
XL	19	257		23	crand	Condition Register AND
XL	19	289		24	creqv	Condition Register Equivalent
XL	19	417		24	crorc	Condition Register OR with Complement
XL	19	449		23	cror	Condition Register OR
XL	19	528	CT	21	bcctr[l]	Branch Conditional to Count Register
M	20		SR	74	rlwimi[.]	Rotate Left Word Immediate then Mask Insert
M	21		SR	71	rlwinm[.]	Rotate Left Word Immediate then AND with Mask
M	23		SR	73	rlwnm[.]	Rotate Left Word then AND with Mask
D	24			64	ori	OR Immediate
D	25			64	oris	OR Immediate Shifted
D	26			64	xori	XOR Immediate
D	27			64	xoris	XOR Immediate Shifted
D	28		SR	63	andi.	AND Immediate
D	29		SR	63	andis.	AND Immediate Shifted
MD	30	0	(SR)	70	rldicl[.]	Rotate Left Doubleword Immediate then Clear Left
MD	30	1	(SR)	70	rldicr[.]	Rotate Left Doubleword Immediate then Clear Right
MD	30	2	(SR)	71	rldic[.]	Rotate Left Doubleword Immediate then Clear
MD	30	3	(SR)	74	rldimi[.]	Rotate Left Doubleword Immediate then Mask Insert
MDS	30	8	(SR)	72	rldcl[.]	Rotate Left Doubleword then Clear Left
MDS	30	9	(SR)	73	rldcr[.]	Rotate Left Doubleword then Clear Right

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
X	31	0		59	cmp	Compare
X	31	4		62	tw	Trap Word
XO	31	8	SR	52	subfc[o][.]	Subtract From Carrying
XO	31	9	(SR)	56	mulhdu[.]	Multiply High Doubleword Unsigned
XO	31	10	SR	52	addc[o][.]	Add Carrying
XO	31	11	SR	56	mulhwu[.]	Multiply High Word Unsigned
X	31	19		81	mfcr	Move From Condition Register
X	31	20		46	lwarx	Load Word And Reserve Indexed
X	31	21	()	35	ldx	Load Doubleword Indexed
X	31	23		33	lwzx	Load Word and Zero Indexed
X	31	24	SR	75	slw[.]	Shift Left Word
X	31	26	SR	68	cntlzw[.]	Count Leading Zeros Word
X	31	27	(SR)	75	sld[.]	Shift Left Doubleword
X	31	28	SR	65	and[.]	AND
X	31	32		60	cmpl	Compare Logical
XO	31	40	SR	51	subf[o][.]	Subtract From
X	31	53	()	35	ldux	Load Doubleword with Update Indexed
X	31	54		Bk II	dcbst	Data Cache Block Store
X	31	55		33	lwzux	Load Word and Zero with Update Indexed
X	31	58	(SR)	68	cntlzd[.]	Count Leading Zeros Doubleword
X	31	60	SR	66	andc[.]	AND with Complement
X	31	68	()	62	td	Trap Doubleword
XO	31	73	(SR)	56	mulhd[.]	Multiply High Doubleword
XO	31	75	SR	56	mulhw[.]	Multiply High Word
X	31	83		Bk III	mfmsr	Move From Machine State Register
X	31	84	()	46	ldarx	Load Doubleword And Reserve Indexed
X	31	86		Bk II	dcbf	Data Cache Block Flush
X	31	87		30	lbzx	Load Byte and Zero Indexed
XO	31	104	SR	54	neg[o][.]	Negate
X	31	119		30	lbzux	Load Byte and Zero with Update Indexed
X	31	124	SR	66	nor[.]	NOR
XO	31	136	SR	53	subfe[o][.]	Subtract From Extended
XO	31	138	SR	53	adde[o][.]	Add Extended
XFX	31	144		81	mtcrf	Move To Condition Register Fields
X	31	146		Bk III	mtmsr	Move To Machine State Register
X	31	149	()	39	stdx	Store Doubleword Indexed
X	31	150		47	stwcx.	Store Word Conditional Indexed
X	31	151		38	stwx	Store Word Indexed
X	31	181	()	39	stdux	Store Doubleword Indexed with Update
X	31	183		38	stwux	Store Word with Update Indexed
XO	31	200	SR	54	subfze[o][.]	Subtract From Zero Extended
XO	31	202	SR	54	addze[o][.]	Add to Zero Extended
X	31	210	{}	Bk III	mtsr	Move To Segment Register
X	31	214	()	47	stdcx.	Store Doubleword Conditional Indexed
X	31	215		36	stbx	Store Byte Indexed
XO	31	232	SR	53	subfme[o][.]	Subtract From Minus One Extended
XO	31	233		55	mulld[o][.]	Multiply Low Doubleword
XO	31	234	SR	53	addme[o][.]	Add to Minus One Extended
XO	31	235		55	mullw[o][.]	Multiply Low Word
X	31	242	{}	Bk III	mtsrin	Move To Segment Register Indirect
X	31	246		Bk II	dcbtst	Data Cache Block Touch for Store
X	31	247		36	stbux	Store Byte with Update Indexed
XO	31	266	SR	51	add[o][.]	Add
X	31	278		Bk II	dcbt	Data Cache Block Touch
X	31	279		31	lhzx	Load Halfword and Zero Indexed
X	31	284	SR	66	eqv[.]	Equivalent
X	31	306		Bk III	tlbie	TLB Invalidate Entry
X	31	310		Bk III	eciwx	External Control In Word Indexed

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
X	31	311		31	lhzux	Load Halfword and Zero with Update Indexed
X	31	316	SR	65	xor[.]	XOR
X	31	338		Bk III	tlbiex	TLB Invalidate Entry by Index
XFX	31	339		80	mfspr	Move From Special Purpose Register
X	31	341	()	34	lwax	Load Word Algebraic Indexed
X	31	343		32	lhax	Load Halfword Algebraic Indexed
X	31	370		Bk III	tlbia	TLB Invalidate All
X	31	371		Bk II	mftb	Move From Time Base
X	31	373	()	34	lwaux	Load Word Algebraic with Update Indexed
X	31	375		32	lhaux	Load Halfword Algebraic with Update Indexed
X	31	407		37	sthx	Store Halfword Indexed
X	31	412	SR	66	orc[.]	OR with Complement
XS	31	413	(SR)	77	sradi[.]	Shift Right Algebraic Doubleword Immediate
X	31	434	()	Bk III	slbie	SLB Invalidate Entry
X	31	438		Bk III	ecowx	External Control Out Word Indexed
X	31	439		37	sthux	Store Halfword with Update Indexed
X	31	444	SR	65	or[.]	OR
XO	31	457	(SR)	58	divdu[o][.]	Divide Doubleword Unsigned
XO	31	459	SR	58	divwu[o][.]	Divide Word Unsigned
X	31	466	()	Bk III	slbiex	SLB Invalidate Entry by Index
XFX	31	467		79	mtspr	Move To Special Purpose Register
X	31	470		Bk III	dcbi	Data Cache Block Invalidate
X	31	476	SR	65	nand[.]	NAND
XO	31	489	(SR)	57	divd[o][.]	Divide Doubleword
XO	31	491	SR	57	divw[o][.]	Divide Word
X	31	498	()	Bk III	slbia	SLB Invalidate All
X	31	512		81	mcrxr	Move to Condition Register from XER
X	31	533		44	lswx	Load String Word Indexed
X	31	534		40	lwbrx	Load Word Byte-Reverse Indexed
X	31	535		101	lfsx	Load Floating-Point Single Indexed
X	31	536	SR	76	srw[.]	Shift Right Word
X	31	539	(SR)	76	srd[.]	Shift Right Doubleword
X	31	566		Bk III	tlbsync	TLB Synchronize
X	31	567		101	lfsux	Load Floating-Point Single with Update Indexed
X	31	595	{}	Bk III	mfsr	Move From Segment Register
X	31	597		44	lswi	Load String Word Immediate
X	31	598		48	sync	Synchronize
X	31	599		102	lfdx	Load Floating-Point Double Indexed
X	31	631		102	lfdux	Load Floating-Point Double with Update Indexed
X	31	659	{}	Bk III	mfsrin	Move From Segment Register Indirect
X	31	661		45	stswx	Store String Word Indexed
X	31	662		41	stwbrx	Store Word Byte-Reverse Indexed
X	31	663		104	stfsx	Store Floating-Point Single Indexed
X	31	695		104	stfsux	Store Floating-Point Single with Update Indexed
X	31	725		45	stswi	Store String Word Immediate
X	31	727		105	stfdx	Store Floating-Point Double Indexed
X	31	759		105	stfdux	Store Floating-Point Double with Update Indexed
X	31	790		40	lhbrx	Load Halfword Byte-Reverse Indexed
X	31	792	SR	78	sraw[.]	Shift Right Algebraic Word
X	31	794	(SR)	78	srad[.]	Shift Right Algebraic Doubleword
X	31	824	SR	77	srawi[.]	Shift Right Algebraic Word Immediate
X	31	854		Bk II	eieio	Enforce In-order Execution of I/O
X	31	918		41	sthbrx	Store Halfword Byte-Reverse Indexed
X	31	922	SR	67	extsh[.]	Extend Sign Halfword
X	31	954	SR	67	extsb[.]	Extend Sign Byte
X	31	982		Bk II	icbi	Instruction Cache Block Invalidate
X	31	983		120	stfiwx	Store Floating-Point as Integer Word Indexed
X	31	986	(SR)	67	extsw[.]	Extend Sign Word

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
X	31	1014		Bk II	dcbz	Data Cache Block set to Zero
D	32			33	lwz	Load Word and Zero
D	33			33	lwzu	Load Word and Zero with Update
D	34			30	lbz	Load Byte and Zero
D	35			30	lbzu	Load Byte and Zero with Update
D	36			38	stw	Store Word
D	37			38	stwu	Store Word with Update
D	38			36	stb	Store Byte
D	39			36	stbu	Store Byte with Update
D	40			31	lhz	Load Halfword and Zero
D	41			31	lhzu	Load Halfword and Zero with Update
D	42			32	lha	Load Halfword Algebraic
D	43			32	lhau	Load Halfword Algebraic with Update
D	44			37	sth	Store Halfword
D	45			37	sthu	Store Halfword with Update
D	46			42	lmw	Load Multiple Word
D	47			42	stmw	Store Multiple Word
D	48			101	lfs	Load Floating-Point Single
D	49			101	lfsu	Load Floating-Point Single with Update
D	50			102	lfd	Load Floating-Point Double
D	51			102	lfdu	Load Floating-Point Double with Update
D	52			104	stfs	Store Floating-Point Single
D	53			104	stfsu	Store Floating-Point Single with Update
D	54			105	stfd	Store Floating-Point Double
D	55			105	stfdu	Store Floating-Point Double with Update
DS	58	0	()	35	ld	Load Doubleword
DS	58	1	()	35	ldu	Load Doubleword with Update
DS	58	2	()	34	lwa	Load Word Algebraic
A	59	18		108	fdivs[.]	Floating Divide Single
A	59	20		107	fsubs[.]	Floating Subtract Single
A	59	21		107	fadds[.]	Floating Add Single
A	59	22		120	fsqrts[.]	Floating Square Root Single
A	59	24		121	fres[.]	Floating Reciprocal Estimate Single
A	59	25		108	fmuls[.]	Floating Multiply Single
A	59	28		109	fmsubs[.]	Floating Multiply-Subtract Single
A	59	29		109	fmadds[.]	Floating Multiply-Add Single
A	59	30		110	fnmsubs[.]	Floating Negative Multiply-Subtract Single
A	59	31		110	fnmadds[.]	Floating Negative Multiply-Add Single
DS	62	0	()	39	std	Store Doubleword
DS	62	1	()	39	stdu	Store Doubleword with Update
X	63	0		115	fcmpu	Floating Compare Unordered
X	63	12		111	frsp[.]	Floating Round to Single-Precision
X	63	14		113	fctiw[.]	Floating Convert To Integer Word
X	63	15		113	fctiwz[.]	Floating Convert To Integer Word with round toward Zero
A	63	18		108	fdiv[.]	Floating Divide
A	63	20		107	fsub[.]	Floating Subtract
A	63	21		107	fadd[.]	Floating Add
A	63	22		120	fsqrt[.]	Floating Square Root
A	63	23		122	fsel[.]	Floating Select
A	63	25		108	fmul[.]	Floating Multiply
A	63	26		121	frsqrte[.]	Floating Reciprocal Square Root Estimate
A	63	28		109	fmsub[.]	Floating Multiply-Subtract
A	63	29		109	fmadd[.]	Floating Multiply-Add
A	63	30		110	fnmsub[.]	Floating Negative Multiply-Subtract
A	63	31		110	fnmadd[.]	Floating Negative Multiply-Add
X	63	32		115	fcmpo	Floating Compare Ordered
X	63	38		118	mtfsb1[.]	Move To FPSCR Bit 1
X	63	40		106	fneg[.]	Floating Negate

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
X	63	64		116	mcrfs	Move to Condition Register from FPSCR
X	63	70		118	mtfsb0[.]	Move To FPSCR Bit 0
X	63	72		106	fmr[.]	Floating Move Register
X	63	134		117	mtfsfi[.]	Move To FPSCR Field Immediate
X	63	136		106	fnabs[.]	Floating Negative Absolute Value
X	63	264		106	fabs[.]	Floating Absolute Value
X	63	583		116	mffs[.]	Move From FPSCR
XFL	63	711		117	mtfsf[.]	Move To FPSCR Fields
X	63	814	()	112	fctid[.]	Floating Convert To Integer Doubleword
X	63	815	()	112	fctidz[.]	Floating Convert To Integer Doubleword with round toward Zero
X	63	846	()	114	fcfid[.]	Floating Convert From Integer Doubleword

[1]See key to mode dependency column, on page 203.

Appendix M. PowerPC Instruction Set Sorted by Mnemonic

This appendix lists all the instructions in the PowerPC Architecture. A page number is shown for instructions that are defined in this Book (Book I, *PowerPC User Instruction Set Architecture*), and the Book number is shown for instructions that are defined in other Books (Book II, *PowerPC Virtual Environment Architecture*, and Book III, *PowerPC Operating Environment Architecture*). If an instruction is defined in more than one Book, the lowest-numbered Book is used.

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
XO	31	266	SR	51	add[o][.]	Add
XO	31	10	SR	52	addc[o][.]	Add Carrying
XO	31	138	SR	53	adde[o][.]	Add Extended
D	14			50	addi	Add Immediate
D	12		SR	51	addic	Add Immediate Carrying
D	13		SR	51	addic.	Add Immediate Carrying and Record
D	15			50	addis	Add Immediate Shifted
XO	31	234	SR	53	addme[o][.]	Add to Minus One Extended
XO	31	202	SR	54	addze[o][.]	Add to Zero Extended
X	31	28	SR	65	and[.]	AND
X	31	60	SR	66	andc[.]	AND with Complement
D	28		SR	63	andi.	AND Immediate
D	29		SR	63	andis.	AND Immediate Shifted
I	18			20	b[l][a]	Branch
B	16		CT	20	bc[l][a]	Branch Conditional
XL	19	528	CT	21	bcctr[l]	Branch Conditional to Count Register
XL	19	16	CT	21	bclr[l]	Branch Conditional to Link Register
X	31	0		59	cmp	Compare
D	11			59	cmpi	Compare Immediate
X	31	32		60	cmpl	Compare Logical
D	10			60	cmpli	Compare Logical Immediate
X	31	58	(SR)	68	cntlzd[.]	Count Leading Zeros Doubleword
X	31	26	SR	68	cntlzw[.]	Count Leading Zeros Word
XL	19	257		23	crand	Condition Register AND
XL	19	129		24	crandc	Condition Register AND with Complement
XL	19	289		24	creqv	Condition Register Equivalent
XL	19	225		23	crnand	Condition Register NAND
XL	19	33		24	crnor	Condition Register NOR
XL	19	449		23	cror	Condition Register OR
XL	19	417		24	crorc	Condition Register OR with Complement
XL	19	193		23	crxor	Condition Register XOR
X	31	86		Bk II	dcbf	Data Cache Block Flush
X	31	470		Bk III	dcbi	Data Cache Block Invalidate
X	31	54		Bk II	dcbst	Data Cache Block Store
X	31	278		Bk II	dcbt	Data Cache Block Touch
X	31	246		Bk II	dcbtst	Data Cache Block Touch for Store
X	31	1014		Bk II	dcbz	Data Cache Block set to Zero
XO	31	489	(SR)	57	divd[o][.]	Divide Doubleword
XO	31	457	(SR)	58	divdu[o][.]	Divide Doubleword Unsigned
XO	31	491	SR	57	divw[o][.]	Divide Word
XO	31	459	SR	58	divwu[o][.]	Divide Word Unsigned

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
X	31	310		Bk III	eciwx	External Control In Word Indexed
X	31	438		Bk III	ecowx	External Control Out Word Indexed
X	31	854		Bk II	eieio	Enforce In-order Execution of I/O
X	31	284	SR	66	eqv[.]	Equivalent
X	31	954	SR	67	extsb[.]	Extend Sign Byte
X	31	922	SR	67	extsh[.]	Extend Sign Halfword
X	31	986	(SR)	67	extsw[.]	Extend Sign Word
X	63	264		106	fabs[.]	Floating Absolute Value
A	63	21		107	fadd[.]	Floating Add
A	59	21		107	fadds[.]	Floating Add Single
X	63	846	()	114	fcfid[.]	Floating Convert From Integer Doubleword
X	63	32		115	fcmpo	Floating Compare Ordered
X	63	0		115	fcmpu	Floating Compare Unordered
X	63	814	()	112	fctid[.]	Floating Convert To Integer Doubleword
X	63	815	()	112	fctidz[.]	Floating Convert To Integer Doubleword with round toward Zero
X	63	14		113	fctiw[.]	Floating Convert To Integer Word
X	63	15		113	fctiwz[.]	Floating Convert To Integer Word with round toward Zero
A	63	18		108	fdiv[.]	Floating Divide
A	59	18		108	fdivs[.]	Floating Divide Single
A	63	29		109	fmadd[.]	Floating Multiply-Add
A	59	29		109	fmadds[.]	Floating Multiply-Add Single
X	63	72		106	fmr[.]	Floating Move Register
A	63	28		109	fmsub[.]	Floating Multiply-Subtract
A	59	28		109	fmsubs[.]	Floating Multiply-Subtract Single
A	63	25		108	fmul[.]	Floating Multiply
A	59	25		108	fmuls[.]	Floating Multiply Single
X	63	136		106	fnabs[.]	Floating Negative Absolute Value
X	63	40		106	fneg[.]	Floating Negate
A	63	31		110	fnmadd[.]	Floating Negative Multiply-Add
A	59	31		110	fnmadds[.]	Floating Negative Multiply-Add Single
A	63	30		110	fnmsub[.]	Floating Negative Multiply-Subtract
A	59	30		110	fnmsubs[.]	Floating Negative Multiply-Subtract Single
A	59	24		121	fres[.]	Floating Reciprocal Estimate Single
X	63	12		111	frsp[.]	Floating Round to Single-Precision
A	63	26		121	frsqrte[.]	Floating Reciprocal Square Root Estimate
A	63	23		122	fsel[.]	Floating Select
A	63	22		120	fsqrt[.]	Floating Square Root
A	59	22		120	fsqrts[.]	Floating Square Root Single
A	63	20		107	fsub[.]	Floating Subtract
A	59	20		107	fsubs[.]	Floating Subtract Single
X	31	982		Bk II	icbi	Instruction Cache Block Invalidate
XL	19	150		Bk II	isync	Instruction Synchronize
D	34			30	lbz	Load Byte and Zero
D	35			30	lbzu	Load Byte and Zero with Update
X	31	119		30	lbzux	Load Byte and Zero with Update Indexed
X	31	87		30	lbzx	Load Byte and Zero Indexed
DS	58	0	()	35	ld	Load Doubleword
X	31	84	()	46	ldarx	Load Doubleword And Reserve Indexed
DS	58	1	()	35	ldu	Load Doubleword with Update
X	31	53	()	35	ldux	Load Doubleword with Update Indexed
X	31	21	()	35	ldx	Load Doubleword Indexed
D	50			102	lfd	Load Floating-Point Double
D	51			102	lfdu	Load Floating-Point Double with Update
X	31	631		102	lfdux	Load Floating-Point Double with Update Indexed
X	31	599		102	lfdx	Load Floating-Point Double Indexed
D	48			101	lfs	Load Floating-Point Single
D	49			101	lfsu	Load Floating-Point Single with Update

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
X	31	567		101	lfsux	Load Floating-Point Single with Update Indexed
X	31	535		101	lfsx	Load Floating-Point Single Indexed
D	42			32	lha	Load Halfword Algebraic
D	43			32	lhau	Load Halfword Algebraic with Update
X	31	375		32	lhaux	Load Halfword Algebraic with Update Indexed
X	31	343		32	lhax	Load Halfword Algebraic Indexed
X	31	790		40	lhbrx	Load Halfword Byte-Reverse Indexed
D	40			31	lhz	Load Halfword and Zero
D	41			31	lhzu	Load Halfword and Zero with Update
X	31	311		31	lhzux	Load Halfword and Zero with Update Indexed
X	31	279		31	lhzx	Load Halfword and Zero Indexed
D	46			42	lmw	Load Multiple Word
X	31	597		44	lswi	Load String Word Immediate
X	31	533		44	lswx	Load String Word Indexed
DS	58	2	()	34	lwa	Load Word Algebraic
X	31	20		46	lwarx	Load Word And Reserve Indexed
X	31	373	()	34	lwaux	Load Word Algebraic with Update Indexed
X	31	341	()	34	lwax	Load Word Algebraic Indexed
X	31	534		40	lwbrx	Load Word Byte-Reverse Indexed
D	32			33	lwz	Load Word and Zero
D	33			33	lwzu	Load Word and Zero with Update
X	31	55		33	lwzux	Load Word and Zero with Update Indexed
X	31	23		33	lwzx	Load Word and Zero Indexed
XL	19	0		25	mcrf	Move Condition Register Field
X	63	64		116	mcrfs	Move to Condition Register from FPSCR
X	31	512		81	mcrxr	Move to Condition Register from XER
X	31	19		81	mfcr	Move From Condition Register
X	63	583		116	mffs[.]	Move From FPSCR
X	31	83		Bk III	mfmsr	Move From Machine State Register
XFX	31	339		80	mfspr	Move From Special Purpose Register
X	31	595	{}	Bk III	mfsr	Move From Segment Register
X	31	659	{}	Bk III	mfsrin	Move From Segment Register Indirect
X	31	371		Bk II	mftb	Move From Time Base
XFX	31	144		81	mtcrf	Move To Condition Register Fields
X	63	70		118	mtfsb0[.]	Move To FPSCR Bit 0
X	63	38		118	mtfsb1[.]	Move To FPSCR Bit 1
XFL	63	711		117	mtfsf[.]	Move To FPSCR Fields
X	63	134		117	mtfsfi[.]	Move To FPSCR Field Immediate
X	31	146		Bk III	mtmsr	Move To Machine State Register
XFX	31	467		79	mtspr	Move To Special Purpose Register
X	31	210	{}	Bk III	mtsr	Move To Segment Register
X	31	242	{}	Bk III	mtsrin	Move To Segment Register Indirect
XO	31	73	(SR)	56	mulhd[.]	Multiply High Doubleword
XO	31	9	(SR)	56	mulhdu[.]	Multiply High Doubleword Unsigned
XO	31	75	SR	56	mulhw[.]	Multiply High Word
XO	31	11	SR	56	mulhwu[.]	Multiply High Word Unsigned
XO	31	233		55	mulld[o][.]	Multiply Low Doubleword
D	7			55	mulli	Multiply Low Immediate
XO	31	235		55	mullw[o][.]	Multiply Low Word
X	31	476	SR	65	nand[.]	NAND
XO	31	104	SR	54	neg[o][.]	Negate
X	31	124	SR	66	nor[.]	NOR
X	31	444	SR	65	or[.]	OR
X	31	412	SR	66	orc[.]	OR with Complement
D	24			64	ori	OR Immediate
D	25			64	oris	OR Immediate Shifted
XL	19	50		Bk III	rfi	Return From Interrupt
MDS	30	8	(SR)	72	rldcl[.]	Rotate Left Doubleword then Clear Left

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
MDS	30	9	(SR)	73	rldcr[.]	Rotate Left Doubleword then Clear Right
MD	30	2	(SR)	71	rldic[.]	Rotate Left Doubleword Immediate then Clear
MD	30	0	(SR)	70	rldicl[.]	Rotate Left Doubleword Immediate then Clear Left
MD	30	1	(SR)	70	rldicr[.]	Rotate Left Doubleword Immediate then Clear Right
MD	30	3	(SR)	74	rldimi[.]	Rotate Left Doubleword Immediate then Mask Insert
M	20		SR	74	rlwimi[.]	Rotate Left Word Immediate then Mask Insert
M	21		SR	71	rlwinm[.]	Rotate Left Word Immediate then AND with Mask
M	23		SR	73	rlwnm[.]	Rotate Left Word then AND with Mask
SC	17	1		22	sc	System Call
X	31	498	()	Bk III	slbia	SLB Invalidate All
X	31	434	()	Bk III	slbie	SLB Invalidate Entry
X	31	466	()	Bk III	slbiex	SLB Invalidate Entry by Index
X	31	27	(SR)	75	sld[.]	Shift Left Doubleword
X	31	24	SR	75	slw[.]	Shift Left Word
X	31	794	(SR)	78	srad[.]	Shift Right Algebraic Doubleword
XS	31	413	(SR)	77	sradi[.]	Shift Right Algebraic Doubleword Immediate
X	31	792	SR	78	sraw[.]	Shift Right Algebraic Word
X	31	824	SR	77	srawi[.]	Shift Right Algebraic Word Immediate
X	31	539	(SR)	76	srd[.]	Shift Right Doubleword
X	31	536	SR	76	srw[.]	Shift Right Word
D	38			36	stb	Store Byte
D	39			36	stbu	Store Byte with Update
X	31	247		36	stbux	Store Byte with Update Indexed
X	31	215		36	stbx	Store Byte Indexed
DS	62	0	()	39	std	Store Doubleword
X	31	214	()	47	stdcx.	Store Doubleword Conditional Indexed
DS	62	1	()	39	stdu	Store Doubleword with Update
X	31	181	()	39	stdux	Store Doubleword Indexed with Update
X	31	149	()	39	stdx	Store Doubleword Indexed
D	54			105	stfd	Store Floating-Point Double
D	55			105	stfdu	Store Floating-Point Double with Update
X	31	759		105	stfdux	Store Floating-Point Double with Update Indexed
X	31	727		105	stfdx	Store Floating-Point Double Indexed
X	31	983		120	stfiwx	Store Floating-Point as Integer Word Indexed
D	52			104	stfs	Store Floating-Point Single
D	53			104	stfsu	Store Floating-Point Single with Update
X	31	695		104	stfsux	Store Floating-Point Single with Update Indexed
X	31	663		104	stfsx	Store Floating-Point Single Indexed
D	44			37	sth	Store Halfword
X	31	918		41	sthbrx	Store Halfword Byte-Reverse Indexed
D	45			37	sthu	Store Halfword with Update
X	31	439		37	sthux	Store Halfword with Update Indexed
X	31	407		37	sthx	Store Halfword Indexed
D	47			42	stmw	Store Multiple Word
X	31	725		45	stswi	Store String Word Immediate
X	31	661		45	stswx	Store String Word Indexed
D	36			38	stw	Store Word
X	31	662		41	stwbrx	Store Word Byte-Reverse Indexed
X	31	150		47	stwcx.	Store Word Conditional Indexed
D	37			38	stwu	Store Word with Update
X	31	183		38	stwux	Store Word with Update Indexed
X	31	151		38	stwx	Store Word Indexed
XO	31	40	SR	51	subf[o][.]	Subtract From
XO	31	8	SR	52	subfc[o][.]	Subtract From Carrying
XO	31	136	SR	53	subfe[o][.]	Subtract From Extended
D	8		SR	52	subfic	Subtract From Immediate Carrying
XO	31	232	SR	53	subfme[o][.]	Subtract From Minus One Extended
XO	31	200	SR	54	subfze[o][.]	Subtract From Zero Extended

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
X	31	598		48	sync	Synchronize
X	31	68	()	62	td	Trap Doubleword
D	2		()	61	tdi	Trap Doubleword Immediate
X	31	370		Bk III	tlbia	TLB Invalidate All
X	31	306		Bk III	tlbie	TLB Invalidate Entry
X	31	338		Bk III	tlbiex	TLB Invalidate Entry by Index
X	31	566		Bk III	tlbsync	TLB Synchronize
X	31	4		62	tw	Trap Word
D	3			61	twi	Trap Word Immediate
X	31	316	SR	65	xor[.]	XOR
D	26			64	xori	XOR Immediate
D	27			64	xoris	XOR Immediate Shifted

[1]Key to Mode Dependency Column

The entry is shown in parentheses () if the instruction is defined only for 64-bit implementations.

The entry is shown in braces {} if the instruction is defined only for 32-bit implementations.

blank The instruction has no mode dependence, except that if the instruction refers to storage when in 32-bit mode, only the low-order 32 bits of the 64-bit effective address are used to address storage. Storage reference instructions include loads, stores, branch instructions, etc.

CT If the instruction tests the Count Register, it tests the low-order 32 bits when in 32-bit mode, and all 64 bits when in 64-bit mode.

SR The instruction's primary function is mode-independent, but the setting of status registers (such as XER and CR0) is mode-dependent.

Index

V

W

X

Z

Last Page - End of Document

PowerPC Virtual Environment Architecture

Book II

Version 1.02

January 8, 1993

Distribution for IBM: softcopy on KISS64

Owner: Jack Kemp
KEMP at AUSVM6
E64S/4A-015
IBM Corporation
Austin, TX 78758
Tele 512-838-1846
Tie Line 678-1846

Technical Content: Ed Silha
silha@austin.ibm.com
E22S/4F-019
IBM Corporation
Austin, TX 78758
Tele 512-838-1848
Tie Line 678-1848

IBM Confidential

NOTES:

- This is a controlled document.
- Verify version and completeness prior to use.
- See the Preface for additional important information.

Preface

This document defines the additional instructions and facilities, beyond those of the PowerPC User Instruction Set Architecture, that are provided by the PowerPC Virtual Environment Architecture. It covers the storage model and related instructions and facilities available to the application programmer, and the Time Base as seen by the application programmer.

Other related documents define the PowerPC User Instruction Set Architecture, the PowerPC Operating Environment Architecture, and PowerPC Implementation Features. Book I, *PowerPC User Instruction Set Architecture* defines the base instruction set and related facilities available to the application programmer. Book III, *PowerPC Operating Environment Architecture* defines the system (privileged) instructions and related facilities. Book IV, *PowerPC Implementation Features* defines the implementation-dependent aspects of a particular implementation.

The PowerPC Architecture consists of the instructions and facilities described in Books I, II, and III. However, the complete description of the PowerPC Architecture as instantiated in a given implementation includes also the material in Book IV for that implementation.

User Responsibilities

- Do not make any unauthorized alterations to the document (user notes permitted).
- Verify the version prior to use. Version verification procedure is described below.
- Verify completeness prior to use. The last page is labeled 'Last Page - End of Document'. The end of the Table of Contents shows the last page number. All pages are numbered sequentially.
- Report any deviations from these procedures to the document owner.

Next Scheduled Review

The next review is expected to be approximately in March, 1993. At least four weeks before this meeting, a DRAFT version of this document will be distributed.

Version Verification for IBM

- Link to the KISS64 disk in Yorktown or a shadow of this disk. In Yorktown, linking to KISS64 can be done with the command "GIME KISS64."
- Browse the newest file with a name of the form "PPC2xxxx LIST3820," by using the "browse" command.
- Verify that your version matches this file.

If your version is not current, please contact the document owner.

Version Verification for Other Firms

To be supplied.

Approval Process

The following procedure is followed for all changes to the content of this document:

- The Power Open Architecture Work Group (PAWG) meets quarterly or more frequently if necessary.
- At least four weeks before a meeting, a version of this document is distributed to the PAWG. It is marked DRAFT. Proposed changes are included and identified with change bars.
- The PAWG meets and decides each issue.
- Final alterations to this document are made, change bars are removed, and the entire document is distributed with a new version number and the word DRAFT removed.
- At the meeting or a subsequent one, new issues are discussed.
- The resulting changes are described in a new version of this document which is derived from the last non-DRAFT version. Proposed changes are identified with change bars, and the document is distributed to the PAWG. This document has a new version number and is marked DRAFT.
- The cycle repeats from the beginning.

Approvals

This version has been approved for user review by the document owner.

Table of Contents

Changes as of 1993/01/08 Version 1.02

change	reason	page
Reworded paragraphs 1 and 2, and deleted paragraph 4 in 1.6.1.1, "The Enforce In-order Execution of I/O Instruction" on page 8.	Agreed at Dec. 2 Power Open meeting.	8
Replaced Engineering Note that said that TLB invalidates must be held off to avoid stuttering, with a sentence in a Programming Note saying that unsynchronized invalidates do not have a defined result.	Agreed at Dec. 2 Power Open meeting.	14
Five of the Data Cache instructions had the paragraph "If EA specifies a storage address for which T=1 (see Book III ...), the instruction is treated as a no-op." Replaced these with a single paragraph at the beginning of the section saying approximately the same thing. Made the same wording change for ***icbi***.	Agreed at Dec. 2 Power Open meeting.	16, 17
Deleted phrase "and need not be constant over long periods of time," and the sentence "The Time Base runs continuously when powered on."	Agreed at Dec. 2 Power Open meeting.	21

Changes as of 1992/10/05

change	reason	page
Stated that cache ops are no-ops in T=1 space.	Clarification requested by B. Dorfman 10/5/92.	16 - 19
Clarified that ***sync*** need not discard prefetched instructions.	This has confused people (***sync*** is not context synchronizing).	8

Changes as of 1992/09/23

change	reason	page
Said that ***eieio*** functions in Write Through (as well as CI) storage, and can be used to prevent access combining operations.	Agreed at Sept. 9 PowerPC meeting.	8, 19
Said that operation of ***dcbst*** and ***dcbf*** is independent of the Write Through and Caching Inhibited/Allowed modes.	Agreed at Sept. 9 PowerPC meeting.	18, 19
Said that if ***larx/stcx.*** addresses Write Through storage, it is implementation-dependent whether it is done correctly or causes a DSI.	Agreed at Sept. 9 PowerPC meeting.	9
Eliminated ***lmd*** and ***stmd*** from the table in Chapter 2, Effect of Operand Placement on Performance.	Agreed at Sept. 9 PowerPC meeting.	13

Changes as of 1992/09/17

change	reason	page
Deleted waiting for TLB invalidate operations on other processors from the list of things that ***sync*** does.	Agreed at Sept. 9 PowerPC meeting.	8
Added that ***sync*** waits for reference and change bits to be updated.	Agreed at Sept. 9 PowerPC meeting.	8
Expanded the functions of ***isync*** to do a context synchronization.	Agreed at Sept. 9 PowerPC meeting.	16
Reworked Time Base section to reflect use of an ***mfspr***-like instruction to read it, and to allow for variable update frequencies.	Agreed at Sept. 9 PowerPC meeting.	21ff

Chapter 1. Storage Model

1.1 Definitions and Notation

The following definitions, in addition to those specified in Book I, are used in this document.

- **main storage**
  The common storage that a processor or other mechanism accesses when it has no cache or has no copy of the storage being accessed in its cache.
- **sequential execution**
  A model for the execution of a sequence of instructions (program) in which one instruction is executed and completed before the next instruction is begun. Instructions are executed in the order in which they appear in the program, except following the execution of a branch instruction, which causes sequential execution to continue at a location specified by the branch instruction.
- **program order**
  The execution of instructions in the strict order in which they occur in the program. See **sequential execution** above.
- **processor**
  A hardware component that executes the PowerPC instructions specified in a program.
- **storage location**
  One or more sequential bytes of storage beginning at the address computed by a storage access instruction. The number of bytes comprising the location depends on the type of storage access instruction being executed.
- **load**
  An instruction that copies one or more bytes from a storage location to one or more registers (GPRs or FPRs).
- **store**
  An instruction that copies one or more bytes from one or more registers (GPR's or FPR's) to a storage location.
- **system**
  A combination of processors, storage, and associated mechanisms that is capable of executing programs. Sometimes the reference to system includes services provided by the operating system.
- **uniprocessor**
  A system that contains one PowerPC processor.
- **multiprocessor**
  A system that contains two or more PowerPC processors.

- **shared storage multiprocessor**
  A multiprocessor that contains some common storage, which all PowerPC processors can access.
- **performed**
  A load is performed with respect to all other processors (and mechanisms) when the value to be returned by the load can no longer be changed by a subsequent store by any processor (or other mechanism).
  A store is performed with respect to all other processors (and mechanisms) when any load from the same location used by the store returns the value stored (or a value stored subsequently).
- **storage page**
  The unit of storage that is managed by the virtual storage system and that can be assigned storage control attributes.

Architecture Note

All processors developed in support of Power Open or MAC-Risc will have a 4096-byte page size.

- **aligned storage access**
  A load or store is aligned if the address of the target storage location is a multiple of the size of the transfer effected by the instruction.
- **atomic access**
  A storage access executed by a processor during which no other processor or mechanism can access any byte of the target location between the time the processor performing the access accesses any byte of the location and the time that it completes the access to all bytes of that location.

1.2 Introduction

The PowerPC User Instruction Set Architecture defines storage as a linear array of bytes indexed from 0 to a maximum of $2^{64} - 1\{2^{32} - 1\}$. Each byte is identified by its index, called its address. Each byte contains a value. This information is sufficient to allow the programming of applications which require no special features of any particular system environment. The PowerPC Virtual Environment Architecture, described herein, expands this simple storage model to include caches, virtual storage, and shared storage multiprocessors. The PowerPC Virtual Environment Architecture in conjunction with services based on the PowerPC Operating Environment Architecture and provided by the operating system permit explicit control of this expanded storage model. A simple model for sequential execution allows at most one storage access to be performed at a time, and requires that all storage accesses appear to be performed in program order. This makes the operation of the model easy to understand and does not require that a program execute any special instructions to guarantee the current state of storage.

The PowerPC architecture specifies a weakly consistent storage model and supports shared storage multiprocessors. In this model, it can be difficult to envision the state of storage at a given instant. When two or more programs or instances of programs share storage, a single program cannot count on the content of a storage location being correct unless it has executed the appropriate synchronization instructions. The features and instructions available in PowerPC systems to enable programs such as these to execute correctly and efficiently are described in this book.

1.3 Memory Coherence

In a PowerPC system, when two or more processors are updating the same storage location, the content of the storage location may not appear to be the same when viewed from different processors at the same instant, nor is the result of stores by two processors to the same location guaranteed to give a predictable result. However, the architecture requires that storage accesses are always performed and that the result of an access by a correct program is never lost.

Coherence refers to the property of the storage subsystem that manages multiple copies of a storage location existing in caches and main storage, and to the manner in which those copies are required to be identical or allowed to be different. As noted in Section 1.4, "Storage Control Attributes" on page 4, the coherence of storage pages may be managed by hardware or software depending on the setting of the Memory Coherence attribute.

Memory coherence is managed in blocks called ***coherence blocks***. Their size is implementation-dependent (see the Book IV, *PowerPC Implementation Features* document for the implementation), but is usually larger than a word and often the size of a cache block.

1.3.1 Coherence Required

When an accessed page is in Memory Coherence Required mode, the processor performing the storage access must participate in a coherence protocol with other processors and the storage subsystem to ensure that updates to a storage location are performed and are not lost. Storage coherence is partly dependent on whether the accesses are atomic, whether they compete, and whether they conflict.

An access is **atomic** if it is always performed in its entirety with no visible fragmentation. Atomic accesses are thus **serialized**: each happens in its entirety in some order, even when that order is not

specified in the program nor enforced between processors.

In PowerPC the following single-register accesses are always atomic:

- byte accesses (all bytes are aligned on byte boundaries)
- halfword accesses aligned on halfword boundaries
- word accesses aligned on word boundaries
- doubleword accesses aligned on doubleword boundaries (64-bit implementations only)

No other accesses are guaranteed to be atomic. In particular, multiple-register loads and stores are not atomic, nor are floating-point doubleword accesses on a 32-bit implementation.

Two accesses **compete** if, in any possible execution, they overlap, there is no order implied between them, and they could be performed simultaneously on different processors. Coherence does not ensure a predictable result when processors access the same location in a conflicting manner. Two competing accesses to the same location **conflict** if at least one is a store. Coherence does ensure predictable results when processors access storage in a manner that does not conflict. The results for several combinations of loads and stores to the same or overlapping locations are described below.

1. When two processors execute atomic stores to locations that do not overlap and no other stores are performed to those locations, the content of those locations is the same as if the two stores were performed by a single processor.
2. When two processors execute atomic stores to the same storage location, and no other store is performed to that location, the content of that location will be the result stored by one of the processors.
3. When two processors execute stores to the same location that are not atomic, and no other store is performed to that location, the result is some combination of the bytes stored by both processors.
4. When two processors execute stores to overlapped locations, and no other store is performed to those locations, the result is some combination of the bytes stored by the processors to the overlapping bytes. The portions of the locations that do not overlap will contain the bytes stored by the processor storing to the location.
5. When a processor executes an atomic store to a location, a second processor executes an atomic load from that location, and no other store is performed to that location, the value returned by the load is the content of the location prior to the store or the content of the location subsequent to the store.
6. When conflicting accesses are not atomic, one access is a load, and no other store is performed to the location, the value returned by the load may be some combination of the content of the location before the store and after the store.

Coherence does not ensure that the result of a store by one processor will be immediately visible to all other processors and mechanisms in the system. Only after a program has executed the ***sync*** instruction are previous storage accesses it executed guaranteed to be globally visible.

1.3.2 Coherence Not Required

When an accessed page is in Memory Coherence Not Required mode, the processor need not enforce storage coherence. This coherence mode may be selected by software to improve performance when it is known that the particular area of storage the processor is accessing will not be accessed by another processor or mechanism. In this mode, software must ensure that the appropriate *Cache Management* instructions have been used to put storage in a consistent state prior to changing the mode or allowing access to that storage area by a different processor or mechanism.

Programming Note

In a single-cache system, Coherence Required is not necessary for correct coherent execution. In fact, in such a system, Coherence Not Required may give better performance.

Engineering Note

If I/O is to be memory coherent, I/O must use the processor's coherence protocol. In such a case, I/O use of the coherence protocol is independent of the setting of the processor's Memory Coherence attribute.

Programming Note

Software must ensure that all locations in a page have been purged from the cache prior to changing the storage mode for the page in such a manner as to restrict the use of the cache (Write Through Not Required to Write Through Required, or Caching Allowed to Caching Inhibited). (See the following section).

1.4 Storage Control Attributes

Some operating systems may provide means to allow programs to specify storage control attributes not described in this document. The definition of these attributes can be found in Book III, *PowerPC Operating Environment Architecture*. The following describes what is expected to be provided when the operating system supports these functions. The details may vary among operating systems, so the details of the specific system being used must be known before these functions can be used.

Generally, the program may use one of each of the following pairs of storage attributes:

- **Write Through Required** or **Not Required**
- **Caching Inhibited** or **Allowed**
- **Memory Coherence Required** or **Not Required**

Not all combinations of these three modes are supported; see Book III, *PowerPC Operating Environment Architecture* for further details.

A program can specify, through an operating system service, the attributes for each page of storage to which it has access. Each load or store will be performed in the following manner, depending on the setting of the storage control attributes for the page of storage containing the addressed storage location.

Write Through

This attribute is meaningful only for Caching Allowed storage. It provides the program control over whether

- the processor is required to update the copy of the storage location in the cache and in main storage, or
- the processor is allowed to update the copy of the storage location in the cache and to defer the update of main storage.

Required

Loads use the copy in the cache if it is there. Stores update the copy of the storage location in the cache if it is in the cache and also update the storage location in main storage.

Not Required

Loads and stores use the copy in the cache if it is there. The block containing the target storage location may be copied to the cache. The storage location in main storage need not contain the value most recently stored to that location.

Caching

Inhibited

When caching is inhibited, the Write Through attribute has no meaning. The load or store is executed in the following manner:

1. The operation is performed to main storage bypassing the cache (i.e., neither the target location nor any of the block(s) containing it are copied into the cache).
2. The operation causes an access (load/store) of appropriate length (i.e., byte, halfword, word, etc.) to the target location in main storage.

It is considered a programming error if a copy of the target location of an access to Caching Inhibited storage is in the cache. Software must ensure that the location has not previously been brought into the cache or, if it has, that it has been flushed from the cache. If the programming error occurs, the result of the access is boundedly undefined.

Allowed

When caching is allowed, the access is performed in the following manner:

1. If the block containing the target storage location is in the cache, it is used.
2. If the block containing the target location is not in the cache, the block(s) of storage containing the target location may be copied to the cache and, if the access is a store, the target location is updated in the cache if it is in the cache.

Memory Coherence

This attribute provides the program control over whether or not the processor maintains storage coherence:

Required

Stores by all processors to the same location are serialized into some order and no processor is able to observe any subset of those stores as occurring in a conflicting order.

Not Required

The order in which one processor observes the stores performed by one or more other processors is undefined.

When coherence is required, its serialization function is effective for all supported combinations of the Write Through and Caching modes (see Book III, *PowerPC Operating Environment Architecture*).

When coherence is not required, the programmer must manage the coherence of storage through use of ***sync*** and *Cache Management* instructions, and facilities provided by the operating system.

1.5 Cache Models

The PowerPC architecture does not require any particular cache organization and allows many different implementations. However, for a program to execute correctly on all implementations, the programmer should assume that separate instruction and data caches exist, and should program to the separate cache model. The functions of these caches are affected by the storage control attributes associated with each storage access as described in 1.4, "Storage Control Attributes" on page 4. *Cache Management* instructions are provided so programs can manage the caches when needed. Depending on the storage control attributes specified by the program and the function being performed, the program may need to use these instructions to guarantee that the function is performed correctly. The *Cache Management* instructions are also useful to optimize the use of memory bandwidth in such applications as graphics and numerically intensive computing.

The processor is not required to maintain copies of storage locations in the instruction cache consistent with changes to storage resulting from the execution of store instructions. Program management of the cache is required when the program generates or modifies code that will be executed (i.e., when the program modifies data storage and then attempts to execute the instructions in the modified storage).

The instructions provided allow the program to

- invalidate the copy of storage in an instruction cache block (***icbi***)
- perform context synchronization, as described in Book III, *PowerPC Operating Environment Architecture* (***isync***)
- copy the content of a data cache block to main storage (***dcbst***)
- copy the content of a data cache block to main storage and make the copy of the block in the data cache invalid (***dcbf***)
- set the content of a data cache block to zeroes (***dcbz***)
- give a hint that a block of storage should be copied into the data cache, so that the copy of the block may be in the cache when subsequent accesses to the block occur, thereby reducing delays (***dcbt, dcbtst***)

The function of the *Cache Management* instructions depends on the implementation of the caches and on the storage control attributes associated with the cache block that is the target of the cache instruction.

There are many variations of cache implementations and the following sections do not attempt to describe them exhaustively. However, the variations that affect the function of the *Cache Management* instructions are discussed here.

Programming Note

Implementations will vary as to what instructions need be executed to perform a function such as code modification. Operating systems are encouraged to provide a service (implementation-dependent) to do the function in an efficient manner.

1.5.1 Split or Dual Caches

Separate caches for instructions and data is called a "Harvard style" cache. This style is the standard PowerPC cache model; that is, it is the model assumed by this architecture and the function of the *Cache Management* instructions depends on this model as well as on the storage control attributes of the target storage block. A copy of a target block in the cache is said to be **marked invalid** if it will not be used for subsequent accesses. The following sections describe the functions performed by each of the *Cache Management* instructions in this model.

1.5.1.1 Instruction Cache Block Invalidate

Invalidating the target block causes any subsequent fetch request for an instruction in the block to not find the block in the cache and to be sent to storage. The instruction performs the following operations:

1. If the target block is not accessible to the program for loads, the system data storage error handler may be invoked.
2. The target block in the instruction cache of the executing processor is marked invalid.
3. If the effective address has an attribute of Coherence Required, the block is invalidated in the instruction caches of all other processors in the system.
4. This access need not be recorded, but if it is, it is considered a load and not a store.

Engineering Note

Causing the system data storage error handler to be invoked if the target block is not accessible to the program for loads facilitates the debugging of software.

1.5.1.2 Data Cache Block Store

This instruction permits the program to ensure that the latest version of the target storage block is in main storage. The instruction performs the following operations:

1. If the target block is not accessible to the program for loads, the system data storage error handler may be invoked.

2. **Memory Coherence**
   Required
   If the target block is in any of the data caches in the system and has been modified, the block is copied to main storage.
   Not Required
   If the target block is in the data cache of the executing processor and has been modified, the block is copied to main storage.
3. This access need not be recorded, but if it is it is considered a load and not a store.

The above action is taken regardless of the setting of the other storage control attributes.

Engineering Note

Causing the system data storage error handler to be invoked if the target block is not accessible to the program for loads facilitates the debugging of software.

1.5.1.3 Data Cache Block Flush

This instruction permits the program to ensure that the latest version of the target storage block is in main storage and no longer in the data cache. The instruction performs the same operations as does the *Data Cache Block Store*. In addition to those operations, the following is done.

Memory Coherence Required
If the target block is in any of the data caches in the system, it is marked invalid in those data caches.
Memory Coherence Not Required
If the target block is in the data cache of the executing processor, it is marked invalid in that data cache.

These actions are taken regardless of the setting of the other storage control attributes.

1.5.1.4 Data Cache Block set to Zero

This instruction permits the program to set large areas of storage to zeros in an efficient manner. The instruction performs the following operations:

1. If the target block is not accessible to the program for stores, the system data storage error handler is invoked.
2. **Caching Inhibited**
   Either each byte of the block in main storage is set to 0x00, or the system alignment error handler is invoked.
3. **Write Through Required**
   Either each byte of the block in main storage is set to 0x00, or the system alignment error handler is invoked.
4. **Memory Coherence**
   - **Required**
     - If the target block is in the data cache of the executing processor, each byte in the block is set to 0x00 and all copies of the block in all data caches are made consistent.
     - If the target block is not in the data cache of the executing processor, the line is established in the data cache without fetching it from storage and each byte in the block is set to 0x00. All copies of the block in all data caches are made consistent.
   - **Not Required**
     - If the target block is in the data cache of the executing processor, each byte in the block is set to 0x00.
     - If the target block is not in the data cache of the executing processor, the line is established in the data cache without fetching it from storage and each byte in the block is set to 0x00.
5. This access must be recorded. It is considered a store to the target location.

1.5.1.5 Data Cache Block Touch

The two *Touch* instructions (one for reading, the other for writing) provide a mechanism by which a program may avoid some of the delays due to accessing storage by attempting to have the target storage location in the cache prior to its first use. These instructions are performance hints and operate as follows:

1. If the target block is not accessible to the program for loads, no other operation is performed.
2. **Caching Inhibited**
   The block is not copied into the cache and no other operations are performed.
3. **Caching Allowed**
   - **Memory Coherence Required**
     If the block is not in the cache, the most recent version of the block may be copied into the cache.
   - **Memory Coherence Not Required**
     If the block is not in the cache, the block may be copied into the cache from main storage without regard for the location of the most recently modified version.
4. This access need not be recorded, but if it is it is considered a load and not a store.

If the instruction is *Touch for Store* and the block is copied into the cache, it is copied in a manner such that a subsequent store to the block will execute efficiently.

The execution of either of these instructions *never* causes the system data error handler to be invoked.

1.5.2 Combined Cache

A combined cache implementation provides a single cache for instructions and data. For this implementation, the *Instruction Cache Block Invalidate* instruction need not perform the same operations as it would for an implementation with separate caches. It can be treated as a no-op, but it is acceptable to invalidate the instruction caches of other processors if the addressed storage is in Coherence Required mode. Following are recommended and required functions of this instruction for combined cache implementations.

Prohibited Operations
It must not invalidate a line in the combined cache that has been modified and the access should not be treated as a store.

Unnecessary Operations
The access should not be treated as a load or store, but to treat it as a load is not a violation of the architecture.

Suggested Operations
If the program executing ***icbi*** does not have access to the target block for loads, the system data storage error handler should be invoked.

1.5.3 Write Through Data Cache

The *Cache Management* instructions affected by the write through implementation are listed in this section. These instructions must perform all the operations specified for a Harvard style cache except as specified in this section. Some of the differences depend on whether the write through implementation is a write through to main storage or just a write through to a second level of cache.

1.5.3.1 Write Through to Main Storage

1. ***Data Cache Block Store***
   By definition, the cache cannot contain a modified block. The processor is not required to copy the target block to main storage.
2. ***Data Cache Block Flush***
   By definition, the cache cannot contain a modified block. The processor is not required to copy the target block to main storage.
3. ***Data Cache Block set to Zero***
   The processor may invoke the system alignment error handler regardless of the setting of the storage control attributes.

1.5.3.2 Write Through to Multi-Level Cache

For *Data Cache Block set to Zero*, the processor may invoke the system alignment error handler regardless of the setting of the storage control attributes.

If a cache is the interface to main storage for all processors and other mechanisms that access storage, that cache can be considered main storage with respect to the *Cache Management* instructions. Otherwise, the cache instructions that cause the content of a cache block to be copied back to main storage or to be marked invalid must be performed against all levels of the cache.

1.6 Shared Storage

This architecture supports the sharing of storage between programs, between different instances of the same program on systems with one or more processors, and between processors and other mechanisms. It also supports access to a storage location by one or more programs using different effective addresses. All these cases are considered storage sharing. Storage is shared in blocks that are an integral number of pages.

When the same storage location has different effective addresses, the addresses are said to be "aliases." Each application can be granted separate access privileges to aliased pages.

Architecture Note

Systems built from processors developed in support of Power Open or MAC-Risc will allow aliasing at the page level. Such systems will accomplish this in a non-architected way.

Engineering Note

Page level aliasing can be implemented in many ways, for example with real addressed caches, L2 directories, or an external signal to an inverse directory. Each processor implementation will decide on its level of implementation in support of its system requirements.

1.6.1 Storage Access Ordering

The PowerPC architecture specifies a **weakly consistent** storage model for shared storage multiprocessor systems. This model provides an opportunity for significantly improved performance over the strongly consistent model, but places the responsibility on the program to ensure that ordering or synchronization instructions are properly placed when necessary for the correct execution of the program.

In this architecture, the order in which the processor performs storage accesses, the order in which those accesses complete in main storage, and the order in which those accesses are viewed as occurring by another processor may all be different. This property is referred to **storage access ordering**. A means of enforcing an ordering of storage accesses is provided to allow programs or instances of programs to share storage. Similar means are needed to allow programs executing on a processor to share storage with some other mechanism such as an I/O device that can also access storage.

The purpose of specifying a weakly consistent storage model is to allow the processor to run very fast for most storage accesses. Two instructions, *Enforce In-order Execution of I/O* and *Synchronize*, are provided that enable the program to control the order in which storage accesses are performed by separate instructions. No ordering should be assumed for the storage accesses done by a multiple-register load or store instruction, and no means are provided for controlling that order.

1.6.1.1 The Enforce In-order Execution of I/O Instruction

The ***eieio*** instruction permits the program to control the order in which *Loads* and *Stores* are performed in main storage. The instruction affects only Caching Inhibited loads and stores, and Write Through Required stores, and only with respect to the order that those accesses complete with respect to main storage. It has no effect on the order that cache accesses are performed.

eieio ensures that all applicable data accesses to main storage previously initiated by the processor have completed with respect to main storage before any applicable storage accesses subsequently initiated by the processor access main storage. It acts like a barrier that flows through the storage queues and to main storage, preventing the reordering of storage accesses across the barrier. The ***eieio*** instruction may complete before previously initiated storage accesses have been performed with respect to other processors and mechanisms.

eieio can be used, for example, to ensure that the data from a sequence of stores to the control registers of an I/O device update those control registers in the order specified by the stores as ordered by ***eieio***.

If stronger ordering is desired or if it is necessary to order accesses to storage that may be in the cache, the ***sync*** instruction must be used.

1.6.1.2 The Synchronize Instruction

When a portion of storage must be forced to a known state, it is necessary to synchronize storage with respect to all processors. This is accomplished by requiring programs to indicate explicitly in the instruction stream that synchronization is required, by inserting a ***sync*** instruction. Only when ***sync*** completes are the effects of all storage accesses executed by the program guaranteed to have been performed with respect to all other processors and mechanisms.

The ***sync*** instruction permits the program to ensure that all storage accesses it has initiated have been performed with respect to all other processors and mechanisms before its next instruction is executed. A program can use this instruction to ensure that all updates to a shared data structure are visible to all other processors prior to executing a store that will release the lock on that data structure. Execution of this instruction does the following:

- Performs the functions described for the ***sync*** instruction in Book I, *PowerPC User Instruction Set Architecture*.
- Ensures that consistency operations, the effects of ***icbi, dcbz, dcbst, dcbf***, and ***dcbi*** (see Book III, *PowerPC Operating Environment Architecture*) executed by the processor executing ***sync*** have completed on all other processors.
- Ensures that TLB invalidates executed by the processor executing ***sync*** have been completed on that processor. However, ***sync*** does *not* wait for such invalidates to be completed on *other* processors (see the Book III section entitled "Table Update Synchronization Requirements").
- Ensures that Reference and Change bits in the Page Table (see Book III, *PowerPC Operating Environment Architecture*) are up to date.

Unlike a context synchronizing operation (see Book III, *PowerPC Operating Environment Architecture*), the ***sync*** instruction need not discard prefetched instructions.

For storage that is maintained as Memory Coherence Not Required, the only effect of ***sync*** on storage operations is to ensure that all previous storage accesses have completed to the level of storage specified by the Caching and Write Through storage control attributes (including the updating of reference and change bits).

Programming Note

The functions performed by ***sync*** will normally take a significant amount of time to complete, so the indiscriminate use of this instruction will adversely affect performance.

1.6.2 Atomic Update Primitives

The *Load and Reserve* and *Store Conditional* instructions together permit atomic update of a storage location. 64-bit implementations have word and doubleword forms of each of these instructions. Described here is the operation of the word forms (***lwarx*** and ***stwcx.***); operation of the doubleword forms (***ldarx*** and ***stdcx.***) is the same except for obvious substitutions.

These instructions function in Caching Inhibited, as well as in Caching Allowed, storage. The addressed page must, however, have the Memory Coherence Required attribute for every processor other than the one doing the atomic update that might execute a store to the location being atomically updated. The remainder of this section assumes that if the system is a multiprocessor, then all processors have the addressed page in Memory Coherence Required mode.

If the addressed storage is in Write Through mode, it is implementation-dependent whether these instructions function correctly or cause the system data storage error handler to be invoked.

The ***lwarx*** is a load from a word-aligned location that has two side effects.

1. A nonspecific reservation for a subsequent ***stwcx.*** or ***stdcx.*** is created.
2. The storage coherence mechanism is notified that a reservation exists for the real address corresponding to the storage location accessed by the ***lwarx***.

The ***stwcx.*** is a store to a word-aligned location that is conditioned on the existence of the reservation created by the ***lwarx*** or ***ldarx***. To emulate an atomic operation with these instructions, it is necessary that both the ***lwarx*** and the ***stwcx.*** access the same storage location even though this requirement is not enforced by the hardware. ***lwarx*** and ***stwcx.*** are ordered by a dependence on the reservation, and the program is not required to insert other instructions to maintain the order of storage accesses by these two instructions.

Engineering Note

Both ***lwarx*** and ***stwcx.*** have a data dependence on the processor reservation resource.

A ***stwcx.*** performs a store to the target storage location only if the storage location accessed by the ***lwarx*** that established the reservation has not been stored into by another processor or mechanism between supplying a value for the ***lwarx*** and storing the value supplied by the ***stwcx.***. In this case, CR0 is set to indicate that the store was performed.

If the ***stwcx.*** completes but does not perform the store because a reservation no longer exists, CR0 is set to indicate that the ***stwcx.*** completed but storage was not altered.

Examples of the use of ***lwarx*** and ***stwcx.*** are given in the *Programming Examples* appendix of Book I, *PowerPC User Instruction Set Architecture*.

When ***stwcx.*** succeeds, its store has been performed but may not yet be globally visible. As a result, a subsequent load or ***lwarx*** on another processor may return a stale value. However, a subsequent ***lwarx*** on the other processor followed by a successful ***stwcx.*** on that processor is guaranteed to have returned the value stored by the first processor's ***stwcx.*** (in the absence of other stores to the location).

Programming Note

To ensure that a store or ***stwcx.*** to a location has become globally visible, it must be followed by a ***sync***. A subsequent load or ***lwarx*** by another processor will then return a value at least as recent as the value stored. This is often more synchronization than is actually needed to ensure program correctness.

1.6.2.1 Reservations

The ability to emulate an atomic operation using ***lwarx*** and ***stwcx.*** is based on the conditional behavior of ***stwcx.***, the ***reservation*** set by ***lwarx***, and the clearing of that reservation if the target location is modified by another processor or other mechanism before the ***stwcx.*** performs.

Programming Note

The combination of ***lwarx*** and ***stwcx.*** improves upon *compare_and_swap* in that the reservation binds the ***lwarx*** and ***stwcx.*** together more reliably. *Compare_and_swap* can only check that the old and current values of the variable are equal, and can cause the program to err if the variable had been modified and the old value subsequently restored. The reservation is always lost if the variable is modified by another processor or mechanism between the ***lwarx*** and ***stwcx.***, so the ***stwcx.*** never succeeds unless the variable has not been stored into (by another processor or mechanism) since the ***lwarx***.

Each processor in a multiprocessor system has at most one reservation at any time. A reservation is established by executing a ***lwarx*** instruction and is lost if any of the following occur:

- The processor holding the reservation issues another ***lwarx*** or ***ldarx***; this clears the first reservation and establishes a new one.
- The processor holding the reservation issues any ***stwcx.*** or ***stdcx.***, whether or not its address matches that of the ***lwarx***.
- Some other processor or other mechanism performs a store in the same reservation granule.

Programming Note

A system error handler may in some cases clear the reservation.

Reservations are not lost under any other circumstances. Specifically, interrupts (see Book III, *PowerPC Operating Environment Architecture*) do not clear reservations (however, system software invoked by interrupts may clear reservations). Immunity to random reservation loss ensures that programs using ***lwarx*** and ***stwcx.*** can make forward progress.

Engineering Note

Reservations must take part in storage coherence. A reservation must be cleared if another processor receives authorization from the coherence mechanism to store to the granule associated with the reservation.

If an implementation continues to hold a reservation when the cache block in which the reservation lies is displaced, the reservation must continue to participate in the coherence protocol. In a snooping implementation, it must join in snooping. In a directory-based implementation, it must register its interest in the reserved line with the directory (shared-read access).

If an implementation demands that the reserved line be held in the cache, it must be able to protect that line from eviction except by cross-invalidates received from other processors as long as the reservation persists. Caches in such an implementation must be sufficiently associative that the machine can continue to run with eviction of the reserved line inhibited.

Programming Note

Programming convention must ensure that ***lwarx*** and ***stwcx.*** addresses match. In proper use, a ***stwcx.*** should be paired with a specific ***lwarx*** to the same real address. Situations in which a ***stwcx.*** may erroneously be issued after some ***lwarx*** other than that with which it is intended to be paired must be scrupulously avoided. For example, there must not be a context change in which the old context leaves a ***lwarx*** dangling and the new context resumes after a ***lwarx*** and before the paired ***stwcx.***. The ***stwcx.*** would be successfully completed, which is not what was intended by the program.

Such a situation must be prevented by issuing a ***stwcx.*** to a dummy writable word-aligned location, as part of the context switch, thereby clearing the reservation of the dangling ***lwarx***. Executing ***stwcx.*** to a word-aligned location suffices to clear the reservation, whether it was obtained by ***lwarx*** or ***ldarx***.

1.6.2.2 Guaranteeing Forward Progress

Forward progress in loops that use ***lwarx*** and ***stwcx.*** is guaranteed by a cooperative effort between hardware, operating system software, and application software. Hardware guarantees that:

- one ***stwcx.*** among a set of processors holding reservations to the same real address will succeed, and
- reservations are not lost unnecessarily, i.e. when the reserved location has not been modified.

While no general rules can be given regarding operating system guarantees, programs that use the examples in the *Programming Examples* appendix of Book I, *PowerPC User Instruction Set Architecture* are guaranteed forward progress.

Architecture Note

The architecture does not include a "fairness algorithm." In competing for a reservation, two processors can indefinitely lock out a third.

1.6.2.3 Reservation Loss Due to Granularity

When one processor holds a reservation, and another processor performs a store that might clear that reservation, the address comparison is done in a way that ignores an implementation-dependent number of low-order bits of the real addresses. The storage block corresponding to the ignored low-order bits is called the **reservation granule**. Its size is implementation-dependent (see the Book IV, *PowerPC Implementation Features* document for the implementation), but is a multiple of the **coherence block** size.

Lock variables should be allocated such that contention for the locks and updates to nearby data structures do not cause excessive reservation losses due to false indications of sharing that can occur due to the reservation granularity.

A processor holding a reservation on the first word of a reservation granule will lose its reservation if some other processor stores elsewhere in that granule. Such problems can be avoided only by ensuring that few such stores occur. This can most easily be accomplished by allocating an entire granule for a lock and wasting all but the first word.

Reservation granularity may vary for each implementation. There are no architectural restrictions bounding the granularity implementations must support, so reasonably portable code must dynamically allocate aligned and padded storage for locks to guarantee absence of granularity-induced conflicts.

1.7 Virtual Storage

The PowerPC system implements a virtual storage model for applications. This means that a combination of hardware and software can present a storage model which allows applications to exist within a "virtual" address space larger than either the effective address space or the real address space.

Each program can access 2^{64} {2^{32}} bytes of "effective address" (EA) space, subject to limitations imposed by the operating system. In a typical PowerPC system, each program's EA space is a subset of a larger "virtual address" (VA) space managed by the operating system.

The operating system is responsible for managing the real (physical) storage resources of the system by means of a "storage mapping" mechanism. Storage is always allocated and managed in units of "pages," which have a fixed, implementation-dependent size. The storage mapping process translates accesses to pages in the EA space into accesses to real pages in main storage.

In general, main storage may not be large enough to contain all of the virtual pages used by the currently active applications. With support provided by hardware mechanisms, the operating system can attempt to use the available real pages to map a sufficient set of effective address pages of the applications. If a sufficient set is maintained, "paging" activity is minimized. If not, performance degradation is likely to occur.

The operating system can support restricted access to pages (including read-write, read-only, and no access), based on system standards (e.g., program code might be read-only) and application requests.

Chapter 2. Effect of Operand Placement on Performance

The placement (location and alignment) of operands in storage will affect relative performance of storage accesses, and in some cases affect it significantly. The best performance is guaranteed if storage operands are aligned. In order to obtain the best performance across the widest range of implementations, the programmer should assume the performance model described in Figure 1 with respect to the placement of storage operands. Performance of accesses varies depending on the following:

1. Operand Size
2. Operand Alignment
3. Crossing no boundary
4. Crossing a Cache Line Boundary
5. Crossing a Page Boundary that is also a protection boundary (see Book III, *PowerPC Operating Environment Architecture*, "Storage Protection").
6. Crossing a BAT Boundary
   See Book III for a description of BAT.
7. Crossing a Segment Boundary
   See Book III for a description of storage segments.

The load/store multiple instructions are defined to operate only on aligned operands. The *Move Assist* instructions have no alignment requirements.

For the purposes of Figure 1, crossing pages with different storage control attributes is equivalent to crossing a segment boundary.

Architecture Note

All processors developed in support of Power Open or MAC-Risc will provide at a minimum the level of support implied by Figure 1.

Page crossing is irrelevant for an access in real mode, within a direct-store segment, and within a BAT area.

Operand		*Boundary Crossing*			
Size	**Byte Align.**	**None**	**Cache Line**	**Page**	**BAT / Seg.**
Integer					
8 Byte	8 4 <4	optimal good poor	– good poor	– poor poor	– poor poor
4 Byte	4 <4	optimal good	– good	– poor	– poor
2 Byte	2 <2	optimal good	– good	– poor	– poor
1 Byte	1	optimal	–	–	–
lmw, stmw	4	good	good	good	poor
string		good	good	poor	poor
Float					
8 Byte	8 4 <4	optimal good poor	– good poor	– poor poor	– poor poor
4 Byte	4 <4	optimal poor	– poor	– poor	– poor

Figure 1. Performance Effects of Storage Operand Placement

2.1 Instruction Restart

If a storage access crosses a page boundary that is also a protection boundary, a BAT boundary, or a segment boundary, a number of conditions could cause the execution of the instruction to be aborted after part of the access has been performed. For example, this may occur when a program attempts to access a page it has not previously accessed, or when the processor must check for a possible change in storage attributes when an access crosses a page boundary. When this occurs, the implementation or the operating system may restart the instruction. If the instruction is restarted, some bytes of the location may be loaded from or stored to the target location a second time.

The following rules apply to storage accesses with regard to restarting the instruction.

Aligned Accesses
A single-register instruction which accesses an aligned operand is never restarted.

Unaligned Accesses
A single-register instruction which accesses an unaligned operand may be restarted if the access crosses a page, BAT, or segment boundary.

Load/Store Multiple, Move Assist
These instructions may be restarted if, in accessing the locations specified by the instruction, a page, BAT, or segment boundary is crossed.

Programming Note

The programmer should assume that any unaligned access in T=0 space might be restarted. Software can ensure this does not occur by use of direct-store or areas covered by BATs (both of which do not have page boundaries).

Unsynchronized TLB invalidates do not have a defined result.

2.2 Atomicity and Order

Access Atomicity
With the exception of double-precision floating-point operands in 32-bit implementations, all aligned accesses are atomic. No other access is required to be atomic. Instructions causing multiple accesses (Load/Store Multiple and *Move Assist*) are not atomic.

Engineering Note

Atomicity of storage accesses is provided by the processor in conjunction with the storage controller. The processor must provide a storage controller interface that is sufficient to allow a storage controller to meet the atomicity requirements specified here.

Access Order
Since the ordering of storage accesses is not guaranteed unless the programmer inserts the appropriate ordering instructions, the order of accesses generated by a single instruction is not guaranteed. Unaligned accesses, Load/Store Multiple instructions, and *Move Assist* instructions have no implicit ordering characteristics. For example, processor A may store a word operand on an odd halfword boundary. It may appear to processor A that the store completed atomically. Processor or other mechanism B, executing a load from the same location, may get a result that is a combination of the value of the first halfword that existed prior to the store by processor A and the value of the second halfword stored by processor A.

Chapter 3. Storage Control Instructions

The instructions in this chapter are not privileged. For most of them, if the applicable cache is not present, the operation is a "no-op" and has no effect on any register or on storage. The only exception is the ***dcbz*** instruction. When the data cache does not exist, ***dcbz*** zeros a certain number of bytes of storage (which has an effect similar to zeroing bytes in a cache block which are later written to storage) or it invokes the system alignment error handler (so its function can be simulated).

As with other storage instructions, the effect of the *Cache Management* instructions on storage is weakly consistent. If the programmer needs to ensure that *Cache Management* or other instructions have been performed with respect to all other processors and mechanisms, a ***sync*** instruction must be placed in the program following those instructions.

The description of many of the *Cache Management* instructions has a statement that defines its storage semantics, such as "This instruction is treated as a store to the addressed byte with respect to address translation and protection." This statement defines the operation of the instruction with respect to how it affects the page reference and change bits, and whether or not interrupts occur for a translation error or a protection violation (see Book III, *PowerPC Operating Environment Architecture*).

Granularity of execution

The maximum allowed cache line size is one page.

The term *block* is used to refer to the amount of storage operated on by each *Cache Management* instruction. The size of a block is not an architectural constant but varies by instruction and by implementation.

3.1 Parameters Useful to Application Programs

It is suggested that the operating system provide a service that allows an application program to obtain the following information.

1. Page size
2. Coherence block size
3. Granule size for reservations
4. An indicator of whether the processor has (a) a combined cache or no caches, or (b) some other cache configuration (split caches or one cache only; if I-cache fetches pass through the D-cache, consider it to be a split cache)
5. Instruction cache total size
6. Data cache total size
7. Instruction cache line size
8. Data cache line size
9. Block size for ***dcbt*** and ***dcbtst*** (if no D-cache, number of bytes zeroed by ***dcbz***)
10. Block size for ***icbi*** (if no I-cache, number of bytes zeroed by ***dcbz***)
11. Block size for ***dcbz***, ***dcbst***, ***dcbf***, and ***dcbi*** (see Book III, *PowerPC Operating Environment Architecture* for a description of ***dcbi***) (if no D-cache, number of bytes zeroed by ***dcbz***)
12. Instruction cache associativity
13. Data cache associativity
14. Factor for converting the Time Base to seconds

If the caches are combined, the same value should be given for an I-cache attribute and the corresponding D-cache attribute.

Architecture Note

All processors in a symmetric multiprocessor must be identical with respect to the cache model, the coherence block size, and the reservation granule size.

3.2 Cache Management Instructions

3.2.1 Instruction Cache Instructions

Instruction caches, if they exist, are not required to be consistent with data caches, storage, nor I/O data transfers. Software must use the appropriate *Cache Management* instructions to ensure that instruction caches are kept consistent when instructions are modified by the processor or by input data transfer. When a processor alters a storage location that may be contained in an instruction cache, software must ensure that updates to storage are visible to the instruction fetching mechanism. Although the instructions to accomplish this vary among implementations and hence many operating systems will provide a system service for this function, the following sequence is typical:

1. ***dcbst*** - update storage
2. ***sync*** - wait for update (see Book I, *PowerPC User Instruction Set Architecture*)
3. ***icbi*** - invalidate copy in instruction cache
4. ***isync*** - perform context synchronization (see Book III, *PowerPC Operating Environment Architecture*)

These operations are necessary because the storage may be in Write Through Not Required mode. Since instruction fetching may bypass the data cache, changes made to items in the data cache may not be reflected in storage until after the instruction fetch completes.

Instruction Cache Block Invalidate X-form

icbi RA,RB

31	///	RA	RB	982	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

If the block containing the byte addressed by EA is in Coherence Required mode, and a block containing the byte addressed by EA is in the instruction cache of any processor, the block is made invalid in all such processors, so that subsequent references cause the block to be refetched.

If the block containing the byte addressed by EA is in Coherence Not Required mode, and a block containing the byte addressed by EA is in the instruction cache of this processor, the block is made invalid in this processor, so that subsequent references cause the block to be fetched from main storage (or perhaps from a data cache).

It is acceptable to treat this instruction as a load from the addressed byte with respect to address translation and protection. Implementations with a combined data and instruction cache may treat the ***icbi*** instruction as a no-op, even to the extent of not validating the EA.

If the EA references storage outside of main storage (see Direct-Store Segments in Book III, *PowerPC Operating Environment Architecture*), the instruction is treated as a no-op.

Special Registers Altered:
None

Engineering Note

It is preferable not to record the storage reference of ***icbi***.

Instruction Synchronize XL-form

isync

[Power mnemonic: ics]

19	///	///	///	150	/
0	6	11	16	21	31

This instruction waits for all previous instructions to complete and then discards any prefetched instructions, causing subsequent instructions to be fetched (or refetched) from storage and to execute in the context established by the previous instructions. This instruction has no effect on other processors or on their caches.

This instruction is context synchronizing (see Book III, *PowerPC Operating Environment Architecture*).

Special Registers Altered:
None

3.2.2 Data Cache Instructions

Data caches and combined caches, if they exist, are required to be consistent with other data caches, combined caches, storage, and I/O data transfers. However, to ensure consistency, aliased effective addresses (two effective addresses that map to the same real address) must have the same page offset (see Section 1.6, "Shared Storage" on page 7).

If the effective address references storage outside of main storage (see Direct-Store Segments in Book III, *PowerPC Operating Environment Architecture*), the instruction is treated as a no-op.

Data Cache Block Touch X-form

dcbt RA,RB

31	///	RA	RB	278	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

This instruction is a hint that performance will probably be improved if the block containing the byte addressed by EA is fetched into the data cache, because the program will probably soon load from the addressed byte. Executing ***dcbt*** will not cause the system error handler to be invoked.

It is acceptable to treat this instruction as a load from the addressed byte with respect to address translation and protection, except that the system error handler must not be invoked for a translation or protection violation.

Special Registers Altered:
None

Programming Note

The purpose of this instruction is to allow the program to request a cache block fetch before it is actually needed by the program. The program can later perform loads to put data into registers. However, the processor is not obliged to load the addressed block into the data cache.

Engineering Note

It is preferable not to record the storage reference of ***dcbt***.

Data Cache Block Touch for Store X-form

dcbtst RA,RB

31	///	RA	RB	246	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

This instruction is a hint that performance will probably be improved if the block containing the byte addressed by EA is fetched into the data cache, because the program will probably soon store into the addressed byte. Executing ***dcbtst*** will not cause the system error handler to be invoked.

It is acceptable to treat this instruction as a load from the addressed byte with respect to address translation and protection, except that the system error handler must not be invoked for a translation or protection violation. Since ***dcbtst*** does not modify storage, it must not be recorded as a store.

Special Registers Altered:
None

Programming Note

The purpose of this instruction is to allow the program to schedule a cache block fetch before it is actually needed by the program. The program can later perform stores to put data into storage. However, the processor is not obliged to load the addressed block into the data cache.

Engineering Note

The *Data Cache Block Touch* instructions are provided for software performance optimization and do not affect the correct execution of a program, regardless of whether they succeed (fetch the target block) or fail (do not fetch the target block).

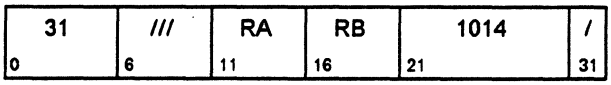
Unlike ***dcbt***, ***dcbtst*** gets exclusive ownership of the line.

It is preferable not to record the storage reference of ***dcbtst***.

Data Cache Block set to Zero X-form

dcbz RA,RB

[Power mnemonic: dclz]



Let the effective address (EA) be the sum (RA|0)+(RB).

If the block containing the byte addressed by EA is in the data cache, all bytes of the block are set to zero.

If the block containing the byte addressed by EA is not in the data cache and the corresponding page is Caching Allowed, the block is established in the data cache without fetching the block from main storage, and all bytes of the block are set to zero.

If the page containing the byte addressed by EA is Caching Inhibited or Write Through, then either (a) all bytes of the area of main storage that corresponds to the addressed block are set to zero, or (b) the system alignment error handler is invoked.

If the block containing the byte addressed by EA is in Coherence Required mode, and the block exists in the data cache(s) of any other processor(s), it is kept coherent in those caches.

This instruction is treated as a store to the addressed byte with respect to address translation and protection.

Special Registers Altered:

None

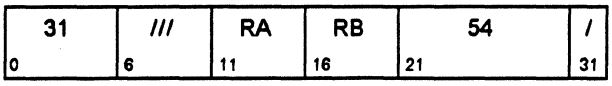
Programming Note

If the page containing the byte addressed by EA is Caching Inhibited or Write Through, the system alignment error handler should set to zero all bytes of the area of main storage that corresponds to the addressed block.

See the *Interrupt* chapter of Book III, *PowerPC Operating Environment Architecture* for a discussion about a possible delayed Machine Check interrupt that can occur by use of ***dcbz*** if the operating system has set up an incorrect storage mapping.

Data Cache Block Store X-form

dcbst RA,RB



Let the effective address (EA) be the sum (RA|0)+(RB).

If the block containing the byte addressed by EA is in Coherence Required mode, and a block containing the byte addressed by EA is in the data cache of any processor and has been modified, the writing of it to main storage is initiated.

If the block containing the byte addressed by EA is in Coherence Not Required mode, and a block containing the byte addressed by EA is in the data cache of this processor and has been modified, the writing of it to main storage is initiated.

The function of this instruction is independent of the Write Through and Caching Inhibited/Allowed modes of the block containing the byte addressed by EA.

It is acceptable to treat this instruction as a load from the addressed byte with respect to address translation and protection.

Special Registers Altered:

None

Engineering Note

It is preferable not to record the storage reference of ***dcbst***.

Data Cache Block Flush X-form

dcbf RA,RB

31	///	RA	RB	86	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The action taken depends on the storage mode associated with the target, and on the state of the block. The list below describes the action taken for the various cases. The actions described must be executed regardless of whether the page containing the addressed byte is in Caching Inhibited or Caching Allowed mode.

1. **Coherence Required**
   Unmodified Block
   Invalidate copies of the block in the caches of all processors.
   Modified Block
   Copy the block to storage. Invalidate copies of the block in the caches of all processors.
   Absent Block
   If modified copies of the block are in the caches of other processors, cause them to be copied to storage and invalidated. If unmodified copies are in the caches of other processors, cause those copies to be invalidated.
2. **Coherence Not Required**
   Unmodified Block
   Invalidate the block in the processor's cache.
   Modified Block
   Copy the block to storage. Invalidate the block in the processor's cache.
   Absent Block
   Do nothing.

The function of this instruction is independent of the Write Through and Caching Inhibited/Allowed modes of the block containing the byte addressed by EA.

It is acceptable to treat this instruction as a load from the addressed byte with respect to address translation and protection.

Special Registers Altered:
None

Engineering Note

It is preferable not to record the storage reference of ***dcbf***.

3.3 Enforce In-order Execution of I/O Instruction

Enforce In-order Execution of I/O X-form

eieio

31	///	///	///	854	/
0	6	11	16	21	31

The ***eieio*** instruction provides an ordering function for the effects of *Load* and *Store* instructions executed by a given processor. Executing an ***eieio*** instruction ensures that all storage accesses previously initiated by the given processor are complete with respect to main storage before any storage accesses subsequently initiated by the given processor access main storage.

eieio orders loads/stores to Caching Inhibited storage and stores to Write Through Required storage. Whether or not it orders accesses to a cache is implementation-dependent.

Special Registers Altered:
None

Programming Note

The ***eieio*** instruction is intended for use only in doing memory-mapped I/O (see Book III, *PowerPC Operating Environment Architecture*) and to prevent load/store combining operations in main storage. It can be thought of as placing a barrier into the stream of storage accesses issued by a processor, such that any given storage access appears to be on the same side of the barrier to both the processor and the I/O device.

The ***eieio*** instruction may complete before previously initiated storage accesses have been performed with respect to other processors and mechanisms.

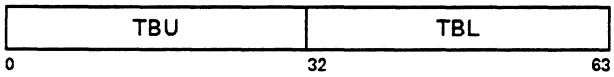
Engineering Note

Unlike the ***sync*** instruction, ***eieio*** need not serialize the processor. ***eieio*** need only ensure that the processor executes storage accesses in the order described above, and enforces that order in any queues in the storage subsystem.

It is permissible to implement ***eieio*** as ***sync***.

Chapter 4. Time Base

The Time Base (TB) is a 64-bit register (see Figure 2) containing a 64-bit unsigned integer which is incremented periodically. Each increment adds 1 to the low-order bit (bit 63). The frequency at which the counter is updated is implementation-dependent.



Field	Description
TBU	Upper 32 bits of Time Base
TBL	Lower 32 bits of Time Base

Figure 2. Time Base

The Time Base increments until its value becomes 0xFFFF_FFFF_FFFF_FFFF ($2^{64} - 1$). At the next increment, its value becomes 0x0000_0000_0000_0000. There is no explicit indication (such as an interrupt) that this has occurred.

The period of the Time Base depends on the driving frequency. As an order of magnitude example, suppose that the CPU clock is 100 MHz and that the Time Base is driven by this frequency divided by 32. Then the period of the Time Base would be

$$T_{TB} = \frac{2^{64} \times 32}{100\ \text{MHz}} = 5.90 \times 10^{12}\ \text{seconds}$$

which is approximately 187,000 years. The PowerPC Architecture does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock, in a PowerPC system. The Time Base update frequency is not required to be constant. What *is* required, so that system software can keep time of day and operate interval timers, is:

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base changes, plus a means to determine what the current update frequency is, ***or***
- The update frequency of the Time Base is under the control of the system software.

Programming Note

Assuming that the operating system initializes the Time Base on power-on to some reasonable value and that the update frequency of the Time Base is constant, the Time Base can be used as a source of values which increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the Time Base will be monotonically increasing. If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

4.1 Time Base Instructions

Extended mnemonics

A pair of extended mnemonics is provided for the ***mftb*** instruction so that it can be coded with the TBR name as part of the mnemonic rather than as a numeric operand. See the Assembler Extended Mnemonics appendix in Book III, *PowerPC Operating Environment Architecture*.

Move From Time Base XFX-form

mftb RT,TBR

31	RT	tbr	371	/
0	6	11	21	31

```
n ← tbr5:9 || tbr0:4
if n = 268 then
  if (64-bit implementation) then
    RT ← TB
  else
    RT ← TB32:63
else if n = 269 then
  if (64-bit implementation) then
    RT ← 320 || TB0:31
  else
    RT ← TB0:31
```

The TBR field denotes either the Time Base or Time Base Upper, encoded as shown in Figure 3. The contents of the designated register are placed into register RT. When reading Time Base Upper on a 64-bit implementation, the high-order 32 bits of register RT are set to zero.

decimal	TBR* $tbr_{5:9}$ $tbr_{0:4}$	Register name	Privileged
268	01000 01100	TB	no
269	01000 01101	TBU	no
* Note that the order of the two 5-bit halves of the TBR number is reversed.			

Figure 3. TBR encodings for mftb

If the TBR field contains any value other than one of the values shown above, the instruction form is invalid.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics for *Move From Time Base*:

Extended:		*Equivalent to:*	
mftb	Rt	mftb	Rt,268
mftbu	Rt	mftb	Rt,269

Programming Note

mftb serves as both a basic and an extended mnemonic. The assembler will recognize an ***mftb*** mnemonic with two operands as the basic form, and an ***mftb*** mnemonic with one operand as the extended form. Another way of saying this is that if ***mftb*** is coded with one operand, then that operand is assumed to be RT, and TBR defaults to the value corresponding to TB.

Compiler and Assembler Note

For the ***mftb*** instruction, the TBR number coded in assembler language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16:20 of the instruction and the low-order 5 bits in bits 11:15.

4.2 Reading the Time Base on 64-bit Implementations

The contents of the Time Base may be read into a GPR by the ***mftb*** extended mnemonic. To read the contents of the Time Base into register Rx, execute:

```
mftb  Rx
```

Reading the Time Base has no effect on the value it contains or the periodic incrementing of that value.

4.3 Reading the Time Base on 32-bit Implementations

On 32-bit implementations, it is not possible to read the entire 64-bit Time Base in a single instruction. The ***mftb*** extended mnemonic moves from the lower half of the Time Base (TBL) to a GPR, and the ***mftbu*** extended mnemonic moves from the upper half (TBU) to a GPR.

Because of the possibility of a carry from TBL to TBU occurring between reads of TBL and TBU, a sequence such as the following is necessary to read the Time Base on 32-bit implementations.

```
loop:
   mftbu  Rx        # load from TBU
   mftb   Ry        # load from TBL
   mftbu  Rz        # load from TBU
   cmpw   Rz,Rx     # see if 'old' = 'new'
   bne    loop      # loop if carry occurred
```

The comparison and loop are necessary to ensure that a consistent pair of values has been obtained.

4.4 Computing Time of Day from the Time Base

Since the update frequency of the Time Base is implementation-dependent, the algorithm for converting the current value in the Time Base to time of day is also implementation-dependent.

As an example, assume that the Time Base is incremented at a constant rate of once for every 32 cycles of a 100 MHZ CPU instruction clock. What is wanted is the pair of 32-bit values comprising a POSIX standard clock: the number of whole seconds which have passed since midnight January 0, 1970, and the remaining fraction of a second expressed as a number of nanoseconds.

Assume that:

- The value 0 in the Time Base represents the start time of the POSIX clock (if this is not true, a simple 64-bit subtraction will make it so).
- Integer constant *ticks_per_sec* contains the value

  $$\frac{100\ \text{MHz}}{32} = 3{,}125{,}000$$

  which is the number of times the Time Base is updated each second.
- Integer constant *ns_adj* contains the value

  $$\frac{1{,}000{,}000{,}000}{3{,}125{,}000} = 320$$

  which is the number of nanoseconds per tick of the Time Base.

64-bit Implementations

The POSIX clock can be computed with an instruction sequence such as this:

```
mftb   Ry           # Ry = Time Base
lwz    Rx,ticks_per_sec
divd   Rz,Ry,Rx     # Rz = whole seconds
stw    Rz,posix_sec
mull   Rz,Rz,Rx     # Rz = quotient * divisor
sub    Rz,Ry,Rz     # Rz = excess ticks
lwz    Rx,ns_adj
mull   Rz,Rz,Rx     # Rz = excess nanoseconds
stw    Rz,posix_ns
```

32-bit Implementations

On a 32-bit machine, direct implementation of the code given above for 64-bit machines is awkward, due mainly to the difficulty of doing 64-bit division.[1] Such division can be avoided entirely if a time of day clock in POSIX format is updated at least once each second.

Assume that:

- The operating system maintains the following variables:
  - *posix_tb* (64 bits)
  - *posix_sec* (32 bits)
  - *posix_ns* (32 bits)

  These variables hold the value of the Time Base and the computed POSIX seconds and nanoseconds values from the last time the POSIX clock was computed.
- The operating system arranges for an interrupt to occur at least once per second, at which time it recomputes the POSIX clock values.
- The integer constant *billion* contains the value 1,000,000,000.

The POSIX clock can be computed with an instruction sequence such as this:

```
loop:
   mftbu  Rx           # Rz = TBU
   mftb   Ry           # Ry = TBL
   mftbu  Rz           # Rz = 'new' TBU value
   cmpw   Rz,Rx        # see if 'old' = 'new'
   bne    loop         # loop if carry occurred
#      now have 64-bit TB in Rx and Ry
   lwz    Rz,posix_tb+4
   sub    Rz,Ry,Rz     # Rz = delta in ticks
   lwz    Rw,ns_adj
   mull   Rz,Rz,Rw     # Rz = delta in ns
   lwz    Rw,posix_ns
   add    Rz,Rz,Rw     # Rz = new ns value
   lwz    Rw,billion
   cmpw   Rz,Rw        # see if past 1 sec
   blt    nochange     # branch if not
   sub    Rz,Rz,Rw     # adjust nanoseconds
   lwz    Rw,posix_sec
   addi   Rw,Rw,1      # adjust seconds
   stw    Rw,posix_sec # store new seconds
nochange:
   stw    Rz,posix_ns  # store new ns
   stw    Rx,posix_tb  # store new time base
   stw    Ry,posix_tb+4
```

Note that the upper part of the Time Base does not participate in the calculation to determine the new POSIX time of day. This is correct as long as the delta value does not exceed one second.

1 See D. E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, section 4.3.1, Algorithm D. Addison-Wesley, 1981.

Non-constant update frequency

In a system in which the update frequency of the Time Base may change over time, it is not possible to convert an isolated Time Base value into time of day. Instead, a Time Base value has meaning only with respect to the current update frequency and the time of day the last time the update frequency was changed. Each time the update frequency changes, the system software is notified of the change via interrupt (or else the change was instigated by the system software itself). At each such change, the system software must compute the current time of day using the old update frequency, compute a new value of **ticks_per_second** for the new frequency, and save the time of day, Time Base value, and tick rate. Subsequent calls to compute time of day use the current Time Base value and the saved data.

Programming Note

A generalized service to compute time of day could take as input

1. Time of day at beginning of current epoch
2. Time Base value at beginning of current epoch
3. Time Base update frequency
4. Time Base value for which time of day is desired

For a PowerPC system in which the Time Base update frequency does not vary, the first three inputs would be constant.

Appendix A. Cross-Reference for Changed Power Mnemonics

The table below lists the Power instruction mnemonics that have been changed in the PowerPC Virtual Environment Architecture, sorted by Power mnemonic.

To determine the PowerPC mnemonic for one of these Power mnemonics, find the Power mnemonic in the second column of the table: the remainder of the line gives the PowerPC mnemonic and the page on which the instruction is described, as well as the instruction names.

Power mnemonics that have not changed are not listed.

Page	Power		PowerPC	
	Mnemonic	**Instruction**	**Mnemonic**	**Instruction**
18	dclz	Data Cache Line Set to Zero	dcbz	Data Cache Block set to Zero
16	ics	Instruction Cache Synchronize	isync	Instruction Synchronize

Appendix B. New Instructions

The following instructions in the PowerPC Virtual Environment Architecture are new: they are not in the Power Architecture. They exist in all PowerPC implementations.

dcbf	Data Cache Block Flush
dcbst	Data Cache Block Store
dcbt	Data Cache Block Touch
dcbtst	Data Cache Block Touch for Store
eieio	Enforce In-order Execution of I/O
icbi	Instruction Cache Block Invalidate
mftb	Move From Time Base

Appendix C. PowerPC Virtual Environment Instruction Set

Form	Opcode		Mode Dep.[1]	Page	Mnemonic	Instruction
	Primary	Extend				
X	31	86		19	dcbf	Data Cache Block Flush
X	31	54		18	dcbst	Data Cache Block Store
X	31	278		17	dcbt	Data Cache Block Touch
X	31	246		17	dcbtst	Data Cache Block Touch for Store
X	31	1014		18	dcbz	Data Cache Block set to Zero
X	31	854		19	eieio	Enforce In-order Execution of I/O
X	31	982		16	icbi	Instruction Cache Block Invalidate
XL	19	150		16	isync	Instruction Synchronize
X	31	371		22	mftb	Move From Time Base

[1]All instructions in the PowerPC Virtual Environment Architecture are mode-independent, except that if the instruction refers to storage when in 32-bit mode, only the low-order 32 bits of the 64-bit effective address are used to address storage.

Index

T

V

W

Last Page - End of Document

PowerPC Operating Environment Architecture

Book III

Version 1.02

January 8, 1993

Distribution for IBM: softcopy on KISS64

Owner: Jack Kemp
KEMP at AUSVM6
E64S/4A-015
IBM Corporation
Austin, TX 78758
Tele 512-838-1846
Tie Line 678-1846

Technical Content: Ed Silha
silha@austin.ibm.com
E22S/4F-019
IBM Corporation
Austin, TX 78758
Tele 512-838-1848
Tie Line 678-1848

IBM Confidential

NOTES:

- This is a controlled document.
- Verify version and completeness prior to use.
- See the Preface for additional important information.

Preface

This document defines the additional instructions and facilities, beyond those of the PowerPC User Instruction Set Architecture and PowerPC Virtual Environment Architecture, that are provided by the PowerPC Operating Environment Architecture. It covers instructions and facilities not available to the application programmer, affecting storage control, interrupts, and timing facilities.

Other related documents define the PowerPC User Instruction Set Architecture, the PowerPC Virtual Environment Architecture, and PowerPC Implementation Features. Book I, *PowerPC User Instruction Set Architecture* defines the base instruction set and related facilities available to the application programmer. Book II, *PowerPC Virtual Environment Architecture* defines the storage model and related instructions and facilities available to the application programmer, and the Time Base as seen by the application programmer. Book IV, *PowerPC Implementation Features* defines the implementation-dependent aspects of a particular implementation.

The PowerPC Architecture consists of the instructions and facilities described in Books I, II, and III. However, the complete description of the PowerPC Architecture as instantiated in a given implementation includes also the material in Book IV for that implementation.

User Responsibilities

- Do not make any unauthorized alterations to the document (user notes permitted).
- Verify the version prior to use. Version verification procedure is described below.
- Verify completeness prior to use. The last page is labeled 'Last Page - End of Document'. The end of the Table of Contents shows the last page number. All pages are numbered sequentially.
- Report any deviations from these procedures to the document owner.

Next Scheduled Review

The next review is expected to be approximately in March, 1993. At least four weeks before this meeting, a DRAFT version of this document will be distributed.

Version Verification for IBM

- Link to the KISS64 disk in Yorktown or a shadow of this disk. In Yorktown, linking to KISS64 can be done with the command "GIME KISS64."
- Browse the newest file with a name of the form "PPC2xxxx LIST3820," by using the "browse" command.
- Verify that your version matches this file.

If your version is not current, please contact the document owner.

Version Verification for Other Firms

To be supplied.

Approval Process

The following procedure is followed for all changes to the content of this document:

- The Power Open Architecture Work Group (PAWG) meets quarterly or more frequently if necessary.
- At least four weeks before a meeting, a version of this document is distributed to the PAWG. It is marked DRAFT. Proposed changes are included and identified with change bars.
- The PAWG meets and decides each issue.
- Final alterations to this document are made, change bars are removed, and the entire document is distributed with a new version number and the word DRAFT removed.
- At the meeting or a subsequent one, new issues are discussed.
- The resulting changes are described in a new version of this document which is derived from the last non-DRAFT version. Proposed changes are identified with change bars, and the document is distributed to the PAWG. This document has a new version number and is marked DRAFT.
- The cycle repeats from the beginning.

Approvals

This version has been approved for user review by the document owner.

Table of Contents

Figures

Changes as of 1993/01/08 Version 1.02

change	reason	page
Delete RTL that shows clearing of the high-order 32 bits of SRR0 and NIA for 64-bit implementations in 32-bit mode.	Redundant and possibly confusing.	9, 10
Simplify second paragraph of ***rfi*** description.	Agreed at Dec. 2 Power Open meeting; a floating-point imprecise interrupt may also be pending.	10
Weaken statement that speculative stores are prohibited.	Agreed at Dec. 2 Power Open meeting.	20
Say that if a load or store will be executed, the entire cache block(s) may be loaded.	Agreed at Dec. 2 Power Open meeting.	20, 41 + 1
Delete Editor's Note about minimum page table size being 2**58 bytes.	Agreed at Dec. 2 Power Open meeting.	29
Delete Programming Notes about ASR, Segment Registers, and SDR 1 re. ***tlbie***.	Agreed at Dec. 2 Power Open meeting.	50
Re. ***tlbsync***, delete "The CPU can be in a multiprocessor system in which other processors have TLBs."	Agreed at Dec. 2 Power Open meeting that it is superfluous.	52
sync required between ***tlbie*** and ***tlbsync***.	Agreed at Dec. 2 Power Open meeting (otherwise ***tlbie*** and ***tlbsync*** could get out of order on the bus).	53 (3 places)
Minor rewording in three Programming Notes.	Agreed at Dec. 2 Power Open meeting.	58 + 1
For process switch, changed reason for ***sync*** from "in case there are data dependences between the processes" to "to ensure that all storage operations of an interrupted process are complete with respect to other processors before that process begins executing on another processor."	Agreed at Dec. 2 Power Open meeting.	58 + 1
For process switch, changed "***isync***" to "***isync*** or ***rfi***."	Agreed at Dec. 2 Power Open meeting.	58 + 1
Say that it is the processor that sets SRR 1 bit 30 to 0 for a nonrecoverable system reset or machine check.	Clarification.	60
Show correct setting of SRR 1 bit 30 for System Reset interrupt.	Agreed at Dec. 2 Power Open meeting (correct an oversight).	60
Add ***eciwx*** and ***ecowx*** to list of instructions that can set $DSISR_5$ for DSI.	Agreed at Dec. 2 Power Open meeting (correct an oversight).	61
Change Programming Note about SRR 0 setting when a pending Imprecise Mode Floating-Point interrupt occurs due to enabling it, to regular text in SRR 0 description.	Agreed at Dec. 2 Power Open meeting. Feeling was that if someone didn't read the note, they might get the architecture wrong.	64
Added phrase "by the time of the next synchronizing event."	Agreed at Dec. 2 Power Open meeting.	64, 67 + 1
Deleted extraneous text.	Text processor error.	67 + 1
Corrected order of operands in ***mftb, mftbu***.	Typo.	76
Clarified that alteration of the V bit is permitted only if the instructions in storage immediately following the ***mtspr*** that alters the IBAT register are also mapped by the segmented address translation mechanism *to the same address, or if the instructions are duplicated in the newly mapped space*.	Agreed at Dec. 2 Power Open meeting.	84 + 1
Said that when updating an IBAT, synchronization is required only if fields in both parts of the IBAT are being altered.	Agreed at Dec. 2 Power Open meeting.	84 + 1
Added ***eciwx, ecowx, stfiwx*** to Alignment/DSISR table.	Omission was an oversight.	90

Changes as of 1992/10/09 Version 1.01 DRAFT

change	reason	page
Noted that System Reset and Machine Check *are* context synchronizing if they are recoverable (i.e., if bit 30 of SRR 1 is set to 1 by the interrupt).	Side effect of adding the RI bit to the MSR.	57, 83

Changes as of 1992/10/05

change	reason	page
Consolidated the various synchronization definitions into one place, namely a new section in the Introduction chapter.	Clarity. Before, context synchronization was defined separately for Branch Processor instructions and for interrupts. And execution synchronization was defined both in the "Definitions and Notation" section and with the ***mtmsr*** instruction.	3
In the new section, stated explicitly that context synchronization requires discarding any prefetched instructions. Also, contrasted the synchronization done by the following: context synchronizing operations, execution synchronizing instructions, and the ***sync*** instruction.	Clarity.	3

Changes as of 1992/10/01

change	reason	page
Said that a processor receiving a ***tlbie/tlbiex*** broadcast will wait for completion of any outstanding storage instructions **including updates to the reference and change bits associated with the invalidated entry.**	PWR_PC FORUM 15:25:57 on 92/09/16, last sentence.	50, 51
Replaced the concept of "volatile" storage with that of "guarded" storage, which is controlled by a "G" bit in the PTE and BAT.	Addendum to PowerPC meeting of 9-11 September 1992.	20ff
Noted that the PR bit of the MSR affects storage protection.	Omission was oversight.	6
Specified how the DAR is set when a Data Storage interrupt occurs on an access to a BAT area.	Omission was oversight.	61
Added fixed-point doubleword load/store that's not word-aligned to the list of potential causes of an Alignment interrupt.	Omission was oversight. Book II allows "poor" performance in this case.	63

Changes as of 1992/09/25

change	reason	page
For imprecise Program interrupt, SRR 0 may point as far as ***sync/isync*** plus four bytes.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	64
Removed interrupt masking function of MSR_{FP}.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	6, 58
Clarified that Branch Trace interrupt is taken whether or not the branch is taken.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	6
Added Arch Note mentioning MSR bits that are used by specific implementations.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	6
Said that some implementations may alter SRR 0/1 for every instruction fetch or data access with IR/DR = 1.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	5, 58
Added a section on mismatched WIM bits.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	42
Said that it's a programming error and results are boundedly undefined if an access is made to CI storage and it's in the cache.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	41
Said that operation of ***dcbi*** is independent of the Write Through and Caching Inhibited/Allowed modes.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	45
Said that load/store combining may be done in CI storage, but that ***eieio*** blocks it in CI and in Write Through stg.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	41
Added text to ***rfi*** definition to specify when pending maskable interrupts are taken after executing the rfi.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	10
Added Eng. Note that in some implementations performance may be improved if I-fetches are done with M=0.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	41
Said that real mode I-fetches may be done with WIM = 000 or 001.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	21
Said that Machine Check will set SRR 1 bit 30 (RI) to 1 if it's not recoverable.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	60
Weakened ***mtmsr*** so that it's execution synchronizing but not context synchronizing.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	15
Defined *execution synchronizing*.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	2
Added that ***lwarx ldarx stwcx. stdcx.*** to Write Through storage may cause a DSI with DSISR bit 5 set.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	61
Eliminated ***lmd*** and ***stmd*** from the discussion of Alignment interrupts and DSISR setting.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	63, 89
Stated that the optional SLB and TLB instructions can be treated as no-ops if the implementation does not have an SLB or TLB. (This is an exception to the general rule that unimplemented optional instructions must cause an Illegal Instruction type Program interrupt.)	As agreed at PowerPC architecture meeting, 9-11 September 1992.	47

change	reason	page
Added unimplemented optional instruction to the list of causes of an Illegal Instruction type Program interrupt.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	64
Updated the appendix on synchronization requirements related to updating any SPR that affects address translation, segment registers, or the MSR.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	83
Specified that the high-order 32 bits of instruction addresses are always 0 in 32-bit mode.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	9
Specified that the high-order 32 bits of SRR 0 and the DAR are always 0 when set by an interrupt from 32-bit mode.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	9

Changes as of 1992/09/18

change	reason	page
Changed Time Base definition such that: update frequency is variable, use ***mtspr*** to write TB and TBU, use new ***mfspr***-like instruction (***mftb***) to read TB and TBU.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	various
Removed requirement that ASR must point to valid segment table when issuing ***slbie*** and that SDR 1 must point to valid page table when issuing ***tlbie*** Allow ***tlbie*** to invalidate or not, broadcast or not, when EA specifies direct-store segment. Added notes to ***tlbiex***, ***tlbia*** regarding what happens when invalidating pages in which another processor is executing.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	48ff
Eliminated PMR.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	various
Expanded Real Address from 52 to 64 bits. Affects PTEs, BATs, and format of SDR 1.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	Chapter 4
Explicitly stated that speculative stores are not permitted.	To fix an oversight; per Rich Oehler.	20
Added concept of "volatile storage," an area in real storage in which speculative storage operations (fetch, load) are not permitted.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	20
Added MSR_{RI}, the "recoverable interrupt" bit, to indicate that state-saving has proceeded far enough that another interrupt (i.e., Machine Check) can be accepted. Set to 0 by hardware on interrupt, set to 1 by software.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	6f
Added WIM = 010 as a supported storage mode.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	42
Explained that ***sync*** does not wait for TLBI's to be completed on other processors.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	53
Added ***tlbsync*** instruction.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	52

change	reason	page
For the Alignment interrupt, added that *lm/stm* crossing a segment or BAT boundary can cause it, and (in the Engineering Note) that it's ok to correctly do the operation.	Correcting "obvious errors."	63
Added initial settings of bits in MSR.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	6f
Relaxed specification of Alignment interrupt's DSISR setting for "don't care" situations.	As agreed at PowerPC architecture meeting, 9-11 September 1992.	63

Chapter 1. Introduction

1.1 Overview

Chapter 1 of Book I, *PowerPC User Instruction Set Architecture* describes computation modes, compatibility with the Power Architecture, document conventions, a general systems overview, instruction formats, and storage addressing. This chapter augments that description as necessary for the PowerPC Operating Environment Architecture.

1.2 Compatibility with the Power Architecture

The PowerPC Architecture provides binary compatibility for Power application programs, except as described in the "Incompatibilities with the Power Architecture" appendix of Book I, *PowerPC User Instruction Set Architecture*. Binary compatibility is not necessarily provided for privileged Power instructions.

1.3 Document Conventions

The notation and terminology used in Book I applies to this document also, with the following substitutions.

- For "system alignment error handler" substitute "Alignment interrupt."
- For "system data storage error handler" substitute "Data Storage interrupt."
- For "system error handler" substitute "interrupt."
- For "system floating-point assist error handler" substitute "Floating-Point Assist interrupt."
- For "system floating-point enabled exception error handler" substitute "Floating-Point Enabled Exception type Program interrupt."
- For "system floating-point unavailable error handler" substitute "Floating-Point Unavailable interrupt."
- For "system illegal instruction error handler" substitute "Illegal Instruction type Program Interrupt."
- For "system instruction storage error handler" substitute "Instruction Storage interrupt."
- For "system privileged instruction error handler" substitute "Privileged Instruction type Program interrupt."
- For "system service program" substitute "System Call interrupt."
- For "system trap handler" substitute "Trap type Program interrupt."

1.3.1 Definitions and Notation

The following augments the definitions given in Book I.

- The context of a program is defined by the content of the MSR when the program is executing. It defines the manner in which the program accesses and executes instructions, accesses data, controls interrupts, accesses the floating-point unit, and interprets addresses or fixed-point data (32 bits or 64 bits).
- An exception is an error, unusual condition, or external signal, that may set a status bit, and which may or may not cause an interrupt, depending upon whether or not the corresponding interrupt is enabled.
- An interrupt is the act of changing the machine state in response to an exception, as described in Chapter 5, "Interrupts" on page 57.
- A trap interrupt is an interrupt that results from execution of a *Trap* instruction.
- Hardware means any combination of hard-wired implementation, "fast trap" to implementation-dependent software assistance, or interrupt for software assistance. In the last case, the interrupt may be to an architected location or to an implementation-dependent location. Any use of fast traps or interrupts to implement the architecture is described in Book IV, *PowerPC Implementation Features*.
- */*, *//*, *///*, ... denotes a field that is reserved in an instruction, a register, or in an architected storage table.

1.3.2 Reserved Fields

System software should initialize reserved fields in architected storage tables (Segment Table, Page Table) to 0s and not keep data in them, as the fields may be used in the future by subsequent versions of PowerPC Architecture.

Some fields of certain storage tables may be written to automatically by hardware, e.g. Reference and Change bits in the Page Table. When the hardware writes to such a table, the following rules must be followed:

- No defined field other than the one(s) the hardware is specifically updating may be modified.
- Contents of reserved fields may be preserved by hardware or such fields may be written as 0s. No other changes to reserved fields may be made.

The handling of reserved bits in status and control registers described in Book I applies here as well. In addition, the reader should be cognizant that reading and writing of some of these registers (e.g., the MSR) can occur as a side effect of processing an interrupt and of returning from an interrupt, as well as when requested explicitly by the appropriate instruction (e.g., ***mtmsr***).

Engineering Note

As noted in Book I, *PowerPC User Instruction Set Architecture*, when a reserved bit in a register is read, the implementation may return either the last value written or the value zero. If all bits of a register are implemented, preserving reserved bits is probably easier. Otherwise, supplying zeros for reserved bits on read (and ignoring them on write) is probably easier.

1.3.3 Description of Instruction Operation

The following augments the definitions given in Book I in the description of the RTL.

Notation	Meaning
SEGREG(x)	Segment Register x

1.4 General Systems Overview

The processor or processor unit contains the sequencing and processing controls for instruction fetch, instruction execution and interrupt action. Instructions that the processing unit can execute fall into a number of classes:

- instructions executed in the Branch Processor
- instructions executed in the Fixed-Point Processor
- instructions executed in the Floating-Point Processor

Almost all instructions executed in the Branch Processor, Fixed-Point Processor, and Floating-Point Processor are non-privileged and are described in Book I, *PowerPC User Instruction Set Architecture*. Book II, *PowerPC Virtual Environment Architecture* contains some cache management instructions. Instructions related to the privileged state of the processor, control of processor resources, control of the storage hierarchy, and all other privileged instructions are described here or in Book IV, *PowerPC Implementation Features*.

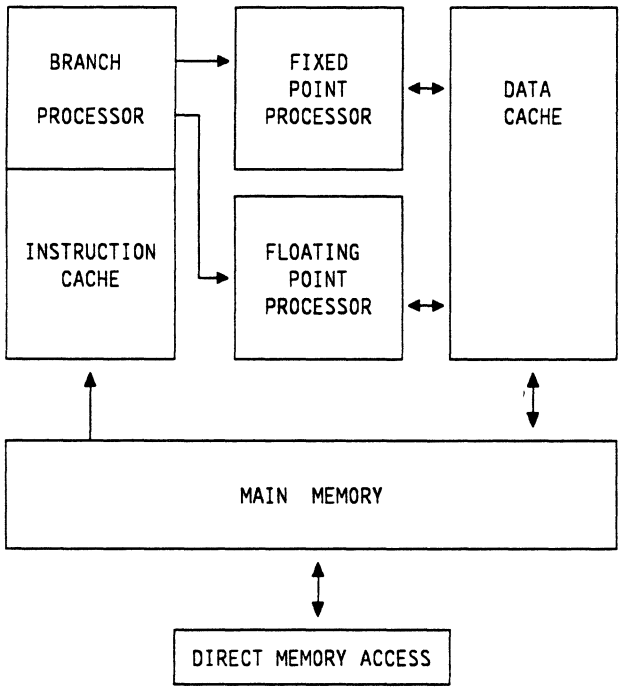


Figure 1. Logical View of the PowerPC Processor Architecture

1.5 Instruction Formats

See Book I, *PowerPC User Instruction Set Architecture* for a description of the instruction formats and addressing.

1.5.1 Instruction Fields

The following augments the instruction fields described in Book I.

SPR (11:20)
Special Purpose Register

See the descriptions of the ***mtspr*** (page 13) and ***mfspr*** (page 14) instructions for a list of SPR encodings.

SR (12:15)
Field used to specify one of the 16 Segment Registers.

1.6 Exceptions

The following augments the list, given in Book I, of exceptions that can be caused by the execution of an instruction.

- the execution of a *Load* or Store instruction to a direct-store segment, in a manner that causes an exception (direct-store error exception)
- the execution of a traced instruction (Trace exception)

1.7 Synchronization

The synchronization described in this section refers to the state of the processor that is performing the synchronization.

1.7.1 Context Synchronization

An instruction or event is "context synchronizing" if it satisfies the requirements listed below. Such instructions and events are collectively called "context synchronizing operations." Examples of context synchronizing operations include the ***rfi*** instruction and most interrupts.

1. The operation causes instruction dispatching (the issuance of instructions by the instruction fetch mechanism to any instruction execution mechanism) to be halted.
2. The operation is not initiated until all instructions already in execution have completed to a point at

which they have reported all exceptions they will cause. (If a storage access due to a previously initiated instruction may cause one or more Direct-Store Error exceptions, the determination of whether it does cause such exceptions is made before the operation is initiated.)

3. If the operation directly causes an interrupt (e.g., ***sc*** directly causes a System Call interrupt) or is an interrupt, the operation is not initiated until no exception exists having higher priority than the exception associated with the interrupt (see Section 5.8, "Interrupt Priorities" on page 67).
4. The instructions that precede the operation will complete execution in the context (privilege, relocation, storage protection, etc.) in which they were initiated.
5. The instructions that follow the operation will be fetched and executed in the context established by the operation. (This requires that any prefetched instructions be discarded, which in turn requires that any effects and side effects of speculatively executing them also be discarded. The only side effects of these instructions that are permitted to survive are those specified in Section 4.2.5, "Speculative Execution" on page 20.)

Unlike the ***sync*** instruction (see Book II, *PowerPC Virtual Environment Architecture*), a context synchronizing operation need not wait for storage-related operations to complete on other processors, nor for Reference and Change bits in the Page Table (see Chapter 4, "Storage Control" on page 17) to be updated.

1.7.2 Execution Synchronization

An instruction is "execution synchronizing" if all previously initiated instructions appear to have completed before the instruction is initiated. An example of an execution synchronizing instruction is ***mtmsr***.

Unlike a context synchronizing operation, an execution synchronizing instruction need not ensure that the instructions following that instruction will execute in the context established by that instruction. This new context becomes effective sometime after the execution synchronizing instruction completes and before or at a subsequent context synchronizing operation.

Chapter 2. Branch Processor

2.1 Branch Processor Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Branch Processor that are in addition to those shown in Book I, *PowerPC User Instruction Set Architecture*.

2.2 Branch Processor Registers

2.2.1 Machine Status Save/Restore Register 0

The Machine Status Save/Restore Register 0 (SRR 0) is a 32-bit or 64-bit register depending on the version of the architecture implemented. This register is used to save machine status on interrupts, and to restore machine status when a Return From Interrupt (*rfi*) instruction is executed.

On interrupt, SRR 0 is set to the current or next instruction address. Thus if the interrupt occurs in 32-bit mode, the high-order 32 bits of SRR 0 are set to 0. When *rfi* is executed, the contents of SRR 0 are copied to the current instruction address (CIA), except that the high-order 32 bits of the CIA are set to 0 when returning to 32-bit mode.

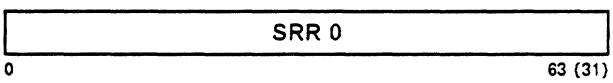


Figure 2. Save/Restore Register 0

In general, SRR 0 contains the instruction address that caused the interrupt, or the instruction address to return to after an interrupt is serviced.

Engineering Note

Since PowerPC instructions must be on word boundaries, the low order 2 bits of SRR 0 need not be implemented. If they are not implemented, these bit positions must return 0 when SRR 0 is read.

Programming Note

In some implementations, every instruction fetch with $MSR_{IR} = 1$, and every load or store with $MSR_{DR} = 1$, may have the side effect of modifying SRR 0.

2.2.2 Machine Status Save/Restore Register 1

The Machine Status Save/Restore Register 1 (SRR 1) is a 32-bit register that is used to save machine status on interrupts, and to restore machine status when an *rfi* instruction is executed.

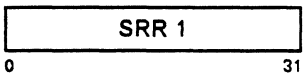


Figure 3. Save/Restore Register 1

In general, when an interrupt occurs, bits 0:15 of SRR 1 are loaded with information specific to the interrupt type, and bits 16:31 of MSR are placed into bits 16:31 of SRR 1.

Programming Note

In some implementations, every instruction fetch with $MSR_{IR}=1$, and every load or store with $MSR_{DR}=1$, may have the side effect of modifying SRR 1.

2.2.3 Machine State Register

The Machine State Register (MSR) is a 32-bit register that defines the state of the processor. On interrupt, the MSR bits are altered in accordance with Figure 30 on page 60. The MSR can also be modified by the ***mtmsr***, **sc**, and ***rfi*** instructions. It can be read by the ***mfmsr*** instruction.

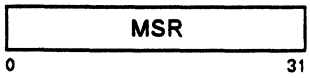


Figure 4. Machine State Register

Below are shown the bit definitions for the Machine State Register.

Bit(s) Description

0:15 Reserved

Architecture Note

Bits 14 and 15 are used by specific implementations, and a proposal is active to use bits 12 and 13 for a specific implementation.

16 ***External Interrupt Enable*** (EE)

0 the processor is disabled against External and Decrementer interrupts.
1 the processor is enabled to take an External or Decrementer interrupt.

17 ***Problem State*** (PR)

0 the processor is privileged to execute any instruction
1 the processor can only execute the non-privileged instructions.

MSR_{PR} also affects storage protection, as described in Chapter 4, "Storage Control" on page 17.

18 ***Floating-Point Available*** (FP)

0 the processor cannot execute any floating-point instructions, including floating-point loads, stores and moves.
1 the processor can execute floating-point instructions.

19 ***Machine Check Enable*** (ME)

0 Machine Check interrupts are disabled.
1 Machine Check interrupts are enabled.

20 ***Floating-Point Exception Mode 0*** (FE0)
See below.

21 ***Single-Step Trace Enable*** (SE)

0 the processor executes instructions normally.
1 the processor generates a Single-Step type Trace interrupt upon the successful execution of the next instruction. Successful execution means the instruction caused no other interrupt. See Book IV, *PowerPC Implementation Features*.

Single-step tracing may not be present on all implementations. If the function is not implemented, MSR_{SE} should be treated as a reserved MSR bit: ***mfmsr*** may return the last value written to the bit, or may return 0 always.

22 ***Branch Trace Enable*** (BE)

0 the processor executes branch instructions normally.
1 the processor generates a Branch type Trace interrupt after completing the execution of a branch instruction, whether or not the branch is taken. See Book IV, *PowerPC Implementation Features*.

Branch tracing may not be present on all implementations. If the function is not implemented, MSR_{BE} should be treated as a reserved MSR bit: ***mfmsr*** may return the last value written to the bit, or may return 0 always.

23 ***Floating-Point Exception Mode 1*** (FE1)
See below.

24 Reserved
This bit corresponds to the AL bit of the Power Architecture. It will not be assigned new meaning in the near future. As for any other reserved bit in a register, software is permitted to write the value 1 to this bit, but there is no guarantee that a subsequent reading of this bit will yield the value that software "wrote" there.

Programming Note

Power-compatible operating systems will probably write the value 1 to this bit.

25 ***Interrupt Prefix*** (IP)
In the following description, **nnnnn** is the offset of the interrupt. See Figure 31 on page 60.

0 interrupts vectored to the real address 0x000n_nnnn in 32-bit versions and real address 0x0000_0000_000n_nnnn in 64-bit versions

1 interrupts vectored to the real address 0xFFFn_nnnn in 32-bit versions and real address 0xFFFF_FFFF_FFFn_nnnn in 64 bit versions.

26 ***Instruction Relocate*** (IR)

0 instruction address translation is off.
1 instruction address translation is on.

27 ***Data Relocate*** (DR)

0 data address translation is off.
1 data address translation is on.

28:29 Reserved

30 ***Recoverable Interrupt*** (RI)

0 interrupt is not recoverable.
1 interrupt is recoverable.

Additional information about the use of this bit is given in Sections 5.4, "Interrupt Processing" on page 58, 5.5.1, "System Reset Interrupt" on page 60, and 5.5.2, "Machine Check Interrupt" on page 60.

31 ***Sixty-Four-bit mode*** (SF) {Reserved}

0 the processor runs in 32-bit mode.
1 the processor runs in 64-bit mode.

Engineering Note

32-bit implementations should ignore attempts to write 1 to MSR_{SF} and should always return 0 when this bit is read.

The Floating-Point Exception Mode bits are interpreted as shown below. For further details see Book I, *PowerPC User Instruction Set Architecture*.

FE0	FE1	Mode
0	**0**	Interrupts disabled
0	**1**	Imprecise Nonrecoverable
1	**0**	Imprecise Recoverable
1	**1**	Precise

Architecture Note

Implementations for use by principal system developers must conform to the following requirements to support system bring-up. The normal sequence of system bring-up is to assert power-on-reset, assert the System Reset interrupt signal, then de-assert power-on-reset. At this time the processor should be able to begin fetching and executing instructions. The initial state of the MSR must be as follows:

Bit	Name	64-bit implementation	32-bit implementation
0:15		unspecified*	unspecified
16	EE	0	0
17	PR	0	0
18	FP	0	0
19	ME	0	0
20	FE0	0	0
21	SE	0	0
22	BE	0	0
23	FE1	0	0
24		unspecified	unspecified
25	IP	1	1
26	IR	0	0
27	DR	0	0
28:29		unspecified	unspecified
30	RI	0	0
31	SF	1	0

* Unspecified, can be 0 or 1

2.2.4 Processor Version Register

The Processor Version Register is a 32-bit read-only register that contains a value identifying the specific version (model) and revision level of the PowerPC processor. The contents of the PVR can be copied to a GPR by the ***mfspr*** instruction. Read access to the PVR is privileged; write access is not provided.

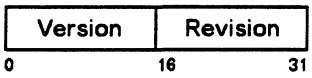


Figure 5. Processor Version Register

The PVR contains two fields:

Version A 16-bit number that uniquely determines a particular processor version and version of the PowerPC Architecture. This number can be used to determine the version of a processor; it may not distinguish between different product models if more than one model uses the same processor.

Revision A 16-bit number that distinguishes between various releases of a particular version, i.e. an Engineering Change level.

The value of the Version portion of the PVR is assigned by the PowerPC Architecture process. Values assigned to date are listed in Appendix E, "Processor Version Numbers" on page 81.

The value of the Revision portion of the PVR is implementation defined.

2.3 Branch Processor Instructions

2.3.1 System Linkage Instructions

These instructions provide the means by which a program can call upon the system to perform a service, and by which the system can return from performing a service or from processing an interrupt.

These instructions are context synchronizing, as defined in Section 1.7.1, "Context Synchronization" on page 3.

The *System Call* instruction is described in Book I, *PowerPC User Instruction Set Architecture*, but only at the level required by an application programmer. A complete description of this instruction appears below.

System Call SC-form

sc

[Power mnemonic: svca]

17	///	///	///	1	/
0	6	11	16	30	31

```
SRR0 ← CIA + 4
SRR1[0:15] ← undefined
SRR1[16:31] ← MSR[16:31]
MSR ← new_value (see below)
NIA ← base_ea + 0xC00 (see below)
```

The effective address of the instruction following the *System Call* instruction is placed into SRR 0. Bits 16:31 of the MSR are placed into bits 16:31 of SRR 1, and bits 0:15 of SRR 1 are set to undefined values.

Then a System Call interrupt is generated. The interrupt causes the MSR to be altered as described in Section 5.5, "Interrupt Definitions" on page 59.

The interrupt causes the next instruction to be fetched from offset 0xC00 from the base real address indicated by the new setting of MSR_{IP}.

This instruction is context synchronizing.

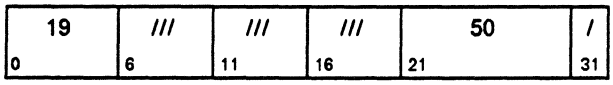
Special Registers Altered:
SRR0 SRR1 MSR

Compatibility Note

For a discussion of Power compatibility with respect to instruction bits 16:29, please refer to the "Incompatibilities with the Power Architecture" appendix of Book I, *PowerPC User Instruction Set Architecture*. For compatibility with future versions of this architecture, these bits should be coded as zero.

Return From Interrupt XL-form

rfi



$MSR_{16:31} \leftarrow SRR1_{16:31}$
$NIA \leftarrow SRR0_{0:61(0:29)} \parallel 0b00$

Bits 16:31 of SRR 1 are placed into bits 16:31 of the MSR. Then the next instruction is fetched, under control of the new MSR value, from the address SRR $0_{0:61(0:29)} \parallel$ 0b00 (32-bit implementations, and 64-bit implementations when SF=1 in the new MSR value) or $^{32}0 \parallel$ SRR $0_{32:61} \parallel$ 0b00 (64-bit implementations when SF=0 in the new MSR value).

If this instruction enables any pending exceptions, the interrupt associated with the highest priority pending exception is generated.

This instruction is privileged and context synchronizing.

Special Registers Altered:
MSR

Chapter 3. Fixed-Point Processor

3.1 Fixed-Point Processor Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Fixed-Point Processor that are in addition to those shown in Book I, *PowerPC User Instruction Set Architecture*.

3.2 PowerPC Special Purpose Registers

The Special Purpose Registers are read and written via the ***mfspr*** (page 14) and ***mtspr*** (page 13) instructions. The descriptions of these instructions list the valid encodings of SPR numbers. Encodings not listed are reserved for future use or for use as implementation-specific registers.

Most SPRs are defined in other parts of this book; see the index to locate those definitions. Some SPRs are specific to an implementation. See Appendix G, "Implementation-Specific SPRs" on page 87 and Book IV, *PowerPC Implementation Features*.

3.3 Fixed-Point Processor Registers

3.3.1 Data Address Register

The Data Address Register (DAR) is a 32-bit or 64-bit register depending on the version of the architecture implemented. See Sections 5.5.3, "Data Storage Interrupt" on page 61, and 5.5.6, "Alignment Interrupt" on page 63.

When an interrupt that uses the DAR occurs, the DAR is set to the effective address associated with the interrupting instruction. If the interrupt occurs in 32-bit mode, the high-order 32 bits of the DAR are set to 0.

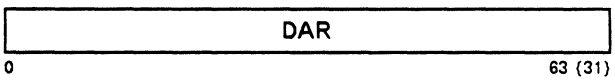


Figure 6. Data Address Register

3.3.2 Data Storage Interrupt Status Register

The Data Storage Interrupt Status Register (DSISR) is a 32-bit register that defines the cause of Data Storage and Alignment interrupts. See Sections 5.5.3, "Data Storage Interrupt" on page 61 and 5.5.6, "Alignment Interrupt" on page 63.

Figure 7. Data Storage Interrupt Status Register

3.3.3 Software-use SPRs

SPRG0 through SPRG3 are 64-bit {32-bit} registers provided for operating system use.

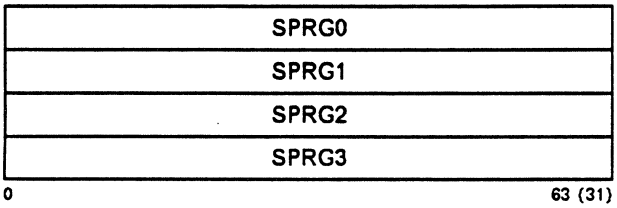


Figure 8. Software-use SPRs

The following list describes the conventional uses of SPRG0 through SPRG3.

SPRG0
Software may load a unique real address in this register to identify an area of storage reserved for use by the first level interrupt handler. This area must be unique for each processor in the system.

SPRG1
This register may be used as a scratch register by the first level interrupt handler to save the content of a GPR. That GPR then can be loaded from SPRG0 and used as a base register to save other GPR's to storage.

SPRG2
This register may be used by the operating system as needed.

SPRG3
This register may be used by the operating system as needed.

3.4 Fixed-Point Processor Privileged Instructions

3.4.1 Move To/From System Registers Instructions

The *Move To Special Purpose Register* and *Move From Special Purpose Register* instructions are described in Book I, *PowerPC User Instruction Set Architecture*, but only at the level available to an application programmer. In particular, no mention is made there of registers that can be accessed only in privileged state. A complete description of these instructions appears below.

Extended mnemonics

A set of extended mnemonics is provided for the ***mtspr*** and ***mfspr*** instructions so that they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand. See Appendix B, "Assembler Extended Mnemonics" on page 75.

Move To Special Purpose Register XFX-form

mtspr SPR,RS

31	RS	spr	467	/
0	6	11	21	31

```
n = spr5:9 || spr0:4
if length(SPREG(n)) = 64 then
  SPREG(n) ← (RS)
else
  SPREG(n) ← (RS)32:63(0:31)
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 9. The contents of register RS are placed into the designated Special Purpose Register. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

$spr_0 = 1$ if and only if writing the register is privileged. Execution of this instruction specifying a defined and privileged register when $MSR_{PR} = 1$ will result in a Privileged Instruction type Program interrupt.

Additional values of the SPR field, beyond those shown in Figure 9, may be defined in Book IV, *PowerPC Implementation Features* for the implementation (see also Appendix G, "Implementation-Specific SPRs" on page 87). If the SPR field contains any value other than one of these implementation-specific values or one of the values shown in the Figure, the instruction form is invalid. For an invalid instruction form in which $spr_0 = 1$, if $MSR_{PR} = 1$ a Privileged Instruction type Program interrupt may occur instead of an Illegal Instruction type Program interrupt.

Special Registers Altered:

See Figure 9

Compiler and Assembler Note

For the ***mtspr*** and ***mfspr*** instructions, the SPR number coded in assembler language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16:20 of the instruction and the low-order 5 bits in bits 11:15. This maintains compatibility with Power SPR encodings, in which these two instructions had only a 5-bit SPR field occupying bits 11:15.

Programming Note

For a discussion of software synchronization requirements when altering certain Special Purpose Registers, please refer to Appendix F, "Synchronization Requirements for Special Registers" on page 83.

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	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
25	00000 11001	SDR 1	yes
26	00000 11010	SRR 0	yes
27	00000 11011	SRR 1	yes
272	01000 10000	SPRG0	yes
273	01000 10001	SPRG1	yes
274	01000 10010	SPRG2	yes
275	01000 10011	SPRG3	yes
280	01000 11000	ASR [2]	yes
282	01000 11010	EAR	yes
284	01000 11100	TB	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

[1] Note that the order of the two 5-bit halves of the SPR number is reversed.

[2] 64-bit implementations only

Figure 9. SPR encodings for mtspr

Compatibility Note

For a discussion of Power compatibility with respect to SPR numbers not shown in the instruction descriptions for ***mtspr*** and ***mfspr***, please refer to the "Incompatibilities with the Power Architecture" appendix of Book I, *PowerPC User Instruction Set Architecture*. For compatibility with future versions of this architecture, only SPR numbers discussed in these instruction descriptions should be used.

Move From Special Purpose Register XFX-form

mfspr RT,SPR

31	RT	spr	339	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
if length(SPREG(n)) = 64 then
  RT ← SPREG(n)
else
  RT ← 32^0 || SPREG(n)
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 10. The contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

$spr_0 = 1$ if and only if reading the register is privileged. Execution of this instruction specifying a defined and privileged register when $MSR_{PR} = 1$ will result in a Privileged Instruction type Program interrupt.

Additional values of the SPR field, beyond those shown in Figure 10, may be defined in Book IV, *PowerPC Implementation Features* for the implementation (see also Appendix G, "Implementation-Specific SPRs" on page 87). If the SPR field contains any value other than one of these implementation-specific values or one of the values shown in the Figure, the instruction form is invalid. For an invalid instruction form in which $spr_0 = 1$, if $MSR_{PR} = 1$ a Privileged Instruction type Program interrupt may occur instead of an Illegal Instruction type Program interrupt.

Special Registers Altered:

None

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	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
25	00000 11001	SDR 1	yes
26	00000 11010	SRR 0	yes
27	00000 11011	SRR 1	yes
272	01000 10000	SPRG0	yes
273	01000 10001	SPRG1	yes
274	01000 10010	SPRG2	yes
275	01000 10011	SPRG3	yes
280	01000 11000	ASR [2]	yes
282	01000 11010	EAR	yes
287	01000 11111	PVR	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

[1] Note that the order of the two 5-bit halves of the SPR number is reversed.

[2] 64-bit implementations only

[3] Moving from the Time Base (TB and TBU) is accomplished with the ***mftb*** instruction, described in Book II.

Figure 10. SPR encodings for mfspr

Compiler/Assembler/Compatibility Notes

See the Notes that appear with ***mtspr***.

Move To Machine State Register X-form

mtmsr RS

31	RS	///	///	146	/
0	6	11	16	21	31

$MSR \leftarrow (RS)_{32:63(0:31)}$

Bits 32:63{0:31} of register RS are placed into the MSR.

This instruction is privileged and execution synchronizing.

In addition, alterations to the EE and RI bits are effective as soon as the instruction completes. Thus if $MSR_{EE}=0$ and an External or Decrementer interrupt is pending, executing an ***mtmsr*** instruction that sets MSR_{EE} to 1 will cause the External or Decrementer interrupt to be taken before the next instruction is executed.

Special Registers Altered:
MSR

Programming Note

For a discussion of software synchronization requirements when altering certain MSR bits, please refer to Appendix F, "Synchronization Requirements for Special Registers" on page 83.

Move From Machine State Register X-form

mfmsr RT

31	RT	///	///	83	/
0	6	11	16	21	31

$RT \leftarrow {}^{32}0\{\} \parallel MSR$

The contents of the MSR are placed into $RT_{32:63(0:31)}$. $RT_{0:31()}$ are set to 0.

This instruction is privileged.

Special Registers Altered:
none

Chapter 4. Storage Control

4.1 Storage Addressing

A program references storage using the Effective Address computed by the processor when it executes a load, store, branch, or cache instruction, and when it fetches the next sequential instruction. The effective address is translated to a real address according to procedures described in section 4.3, "Address Translation Overview" on page 22 and following. The real address is what is sent to the memory subsystem. See Figure 11 on page 22.

For a complete discussion of storage addressing and effective address calculation, refer to "Storage Addressing" in Chapter 1 of Book I, *PowerPC User Instruction Set Architecture*.

Storage Control Overview

- Page size is 2^{12} bytes (4 KB)
- Segment size is 2^{28} bytes (256 MB)
- For 64-bit implementations:
  - Maximum real memory size 2^{64} bytes (16 EB)
  - Effective Address Range 2^{64}
  - Virtual Address Range 2^{80}
  - Number of segments 2^{52}
- For 32-bit implementations:
  - Maximum real memory size 2^{32} bytes (4 GB)
  - Effective Address Range 2^{32}
  - Virtual Address Range 2^{52}
  - Number of segments 2^{24}
- Two types of storage segments based on the state of the T bit in the Segment Table Entry or segment register selected by the Effective Address:
  - T=0: Ordinary storage segment
  - T=1: Direct-store segment

4.2 Storage Model

The storage model provides the following features:

1. The architecture allows the storage implementations to take advantage of the performance benefits of weak ordering of storage access between processors or between processors and devices.
2. The architecture provides instructions that allow the programmer to ensure a consistent and ordered storage state.

   - ***dcbf***
   - ***dcbst***
   - ***dcbz***
   - ***icbi***
   - ***isync***
   - ***ldarx***
   - ***lwarx***
   - ***eieio***
   - ***stdcx.***
   - ***stwcx.***
   - ***sync***

3. Processor ordering: storage accesses by a single processor appear to complete sequentially from the view of the programming model but may complete out of order with respect to the ultimate destination in the storage hierarchy. Order is guaranteed at each level of the storage hierarchy for accesses to the same address from the same processor.
4. Storage consistency between processors and between a processor and I/O is controlled by software through mode bits in the page table. See 4.8.2, "Supported Storage Modes" on page 42. Six modes are supported using the control bits:
   - write through
   - caching inhibited
   - memory coherence

Engineering Note

The architecture does not suggest or preclude any implementation of storage consistency supporting the features listed above. In particular, the implementation may be a snoopy bus design, a centralized cache directory design, or other design.

4.2.1 Storage Segments

Storage is divided into 256 MB (2^{28}) segments.

Programming Note

It is possible to provide larger segments to application programs by using multiple adjacent segments.

These segments can be of two types:

- An *ordinary storage segment*, referred to as a "storage segment" or simply as a "segment." Address translation is controlled by the setting of the relocate bits MSR_{DR} for data and MSR_{IR} for

instructions. MSR_{IR} and MSR_{DR} are independent bits and may be set differently. The state of these bits may be changed by interrupts or by executing the appropriate instructions. An effective address in these segments represents a real or virtual address depending on the setting of the relocate bits of the MSR.

- A *direct-store segment*, always referred to by the entire name "direct-store segment." Such segments may be used for access to I/O. Instruction fetch from direct-store segments is not allowed. MSR_{DR} must be 1 when accessing data in a direct-store segment. See 4.6, "Direct-Store Segments" on page 37 for an explanation of direct-store segments.

The value of the T bit in the Segment Table Entry or Segment Register distinguishes between ordinary storage segments and direct-store segments.

T	*Segment type*
0	Ordinary storage segment
1	Direct-store segment

The T bit in the Segment Table Entry or Segment Register is ignored when fetching instructions with $MSR_{IR}=0$ or when accessing data with $MSR_{DR}=0$. Such accesses are not considered references to direct-store segments.

See also section 4.6, "Direct-Store Segments" on page 37.

4.2.2 Storage Exceptions

Each Effective Address must be translated to real in order to complete the storage access. A *storage exception* occurs if this translation fails for one of the following reasons:

64-bit implementations

- There is no valid entry in the Segment Table for the segment specified by the Effective Address.
- The appropriate Segment Table entry is found, but there is no valid entry in the Page Table for the page specified by the Effective Address.
- Both the appropriate Segment Table and Page Table entries are found, but the access is not allowed by the storage protection mechanism.

32-bit implementations

- There is no valid entry in the Page Table for the page specified by the Effective Address.
- The appropriate Page Table entry is found but the access is not allowed by the storage protection mechanism.

Storage exceptions cause Instruction Storage interrupts and Data Storage interrupts that identify the address of the failing instruction.

In certain cases a storage exception may result in the "restart" of (re-execution of at least part of) a load or store instruction. See the section entitled "Instruction Restart" in Book II, *PowerPC Virtual Environment Architecture*

4.2.3 Instruction Fetch

Instructions are fetched under control of MSR_{IR}. When any context synchronizing event occurs, any prefetched instructions are discarded, and then refetched using the then-current state of MSR_{IR}.

$MSR_{IR}=0$

When instruction relocation is off, $MSR_{IR}=0$, the effective address is interpreted as described in section 4.2.6, "Real Addressing Mode" on page 21.

$MSR_{IR}=1$

Instructions are fetched using the address translated by one of the following mechanisms:

1. Segmented Address Translation Mechanism
2. Block Address Translation Mechanism

Instruction fetch from direct-store segments is not supported. An attempt to execute an instruction fetched from a direct-store segment will result in an Instruction Storage interrupt.

4.2.4 Data Storage Access

Data accesses are controlled by MSR_{DR}. When the state of MSR_{DR} changes, subsequent accesses are made using the new state of MSR_{DR}.

$MSR_{DR}=0$

When data relocation is off, $MSR_{DR}=0$, the effective address is interpreted as described in section 4.2.6, "Real Addressing Mode" on page 21.

$MSR_{DR}=1$

When address relocation is on, $MSR_{DR}=1$, the effective address is translated by one of the following mechanisms:

1. Segmented Address Translation Mechanism
2. Block Address Translation Mechanism
3. Direct-Store Segment Translation Mechanism

4.2.5 Speculative Execution

Data Access

A **speculative operation** is one that a program "might" perform and that the hardware decides to execute out of order on the *speculation* that the result will be needed. If subsequent events indicate that the speculative instruction would not have been executed, the processor abandons any result the instruction produced. Typically, hardware executes instructions speculatively when it has resources that would otherwise be idle, so that the operation is done without cost or almost so.

Most operations can be performed speculatively, as long as the machine appears to follow a simple sequential model such as presented in Book I, *PowerPC User Instruction Set Architecture*. Certain speculative operations are not permitted:

- A speculative store may not be performed in such a manner that the alteration of the target location can be observed by other processors or mechanisms until it can be determined that the store is no longer speculative.
- Speculative loads from direct-store segments (T = 1) are prohibited.
- Speculative loads from "guarded storage" (see below) are prohibited, except that if a load or store operation will be executed, the entire cache block(s) containing the referenced data may be loaded into the cache.
- No error of any kind other than Machine Check may be reported due to the speculative execution of an instruction, until such time as it is known that execution of the instruction is required.

Speculative loads are allowed from any storage that is not "guarded storage." If in doing so a Machine Check exception results, a Machine Check interrupt may be generated even though the data access that caused the Machine Check exception would not have been performed because a previous uncompleted operation would have changed the execution path.

Only one side effect (other than Machine Check) of speculative execution is permitted when a speculative instruction's result is abandoned: the Reference bit in a Page Table Entry may be set due to a speculative load.

Engineering Note

While speculative execution of the storage synchronization instructions (***lwarx***, ***ldarx***, ***stwcx.***, and ***stdcx.***) is permitted by PowerPC architecture, doing so is extremely complex and should be avoided.

Instruction Prefetch

The processor typically fetches instructions ahead of the one(s) currently being executed in order to avoid delay. Such **instruction prefetching** is a speculative operation in that prefetched instructions may not be executed due to intervening branches or interrupts.

Most prefetching is permitted, as long as the machine appears to follow a simple sequential model such as presented in Book I, *PowerPC User Instruction Set Architecture*. Certain prefetching is not permitted:

- Neither fetching nor prefetching from direct-store segments (T = 1) is permitted.
- Prefetching from "guarded storage" (see below) is prohibited, except that if an instruction in a cache block will be executed, the entire cache block may be loaded.
- No error of any kind other than Machine Check may be reported due to instruction prefetching, until such time as the instruction that is the target of such prefetch becomes the instruction to be executed.

Speculative instruction fetches are allowed from any storage that is not "guarded storage." If in doing so, a Machine Check exception results, a Machine Check interrupt may be generated even if the instruction fetch that caused the Machine Check exception would not have been executed because a previous uncompleted operation would have changed the execution path.

Only one side effect (other than Machine Check) of instruction prefetching is permitted: the Reference bit in a Page Table Entry may be set.

Guarded Storage

Storage is said to be "guarded" if either (a) the G bit is one in the relevant PTE or BAT register, or (b) MSR bit IR or DR is zero for instruction fetches or data loads respectively. (In case (b) all of storage is guarded).

Storage in a guarded area may not be well-behaved with regard to prefetching and other speculative storage operations. Such storage may represent an I/O device, and a speculative load or instruction fetch directed to such a device may cause the device to perform unexpected or incorrect operations.

Storage addresses in a guarded area may not have successors; that is, there may be "holes" in a guarded area of the real address space. On any system, the highest real address has no successor. Lack of a successor address means that speculative sequential operations such as instruction prefetching may fail and may result in a Machine Check.

Because of the unpredictable nature of storage in a guarded area, instruction prefetching and speculative loads are not permitted in a guarded area unless the target location is already in the cache. Instruction prefetching in a guarded area is, however, permitted to the extent that if any instruction in a cache block will be executed, the entire cache block containing that instruction may be prefetched into the cache (and instruction buffer). In a similar manner, if a load or store operation will be executed, the entire cache block(s) containing the referenced data may be loaded into the cache.

4.2.6 Real Addressing Mode

Whether address translation is enabled is controlled by MSR_{IR} for instruction fetching and by MSR_{DR} for data loads and stores. If address translation is disabled for a particular access (fetch, load, or store), the Effective Address is treated as the Real Address and is passed directly to the memory subsystem.

The EA is a 64-bit {32-bit} quantity computed by the CPU. The width of the Real Address supported by a particular implementation will be less than or equal to this. If it is less, the high-order bits of the EA are **ignored** when forming the Real Address.

Accesses in real mode bypass all storage protection checks (section 4.10) and do not cause the recording of reference and change information (section 4.9). Real mode data accesses are executed as though the storage access mode bits "WIMG" were 0011 (section 4.8). This mode allows accesses to be cached, does not require the accesses to be written through the cache to main storage, requires the hardware to enforce data consistence with storage, I/O, and other processors (caches), and treats all storage as guarded storage. Real mode instruction fetches are executed as though the "WIMG" bits were either 0001 or 0011. Speculative fetching of instructions and speculative loads from storage in real mode are prohibited (see "Guarded Storage" above).

Access to direct-store segments (section 4.6) is not possible when translation is disabled, as Segment Table Entries (section 4.4.1.2) or Segment Registers (section 4.5.1.1) are not checked for a T=1 specification.

WARNING: An attempt to fetch from, load from, or store to a Real Address that is not physically present in the machine may result in a Machine Check interrupt or a Checkstop (Section 5.5.2).

4.3 Address Translation Overview

Figure 11 gives an overview of the address translation process on PowerPC.

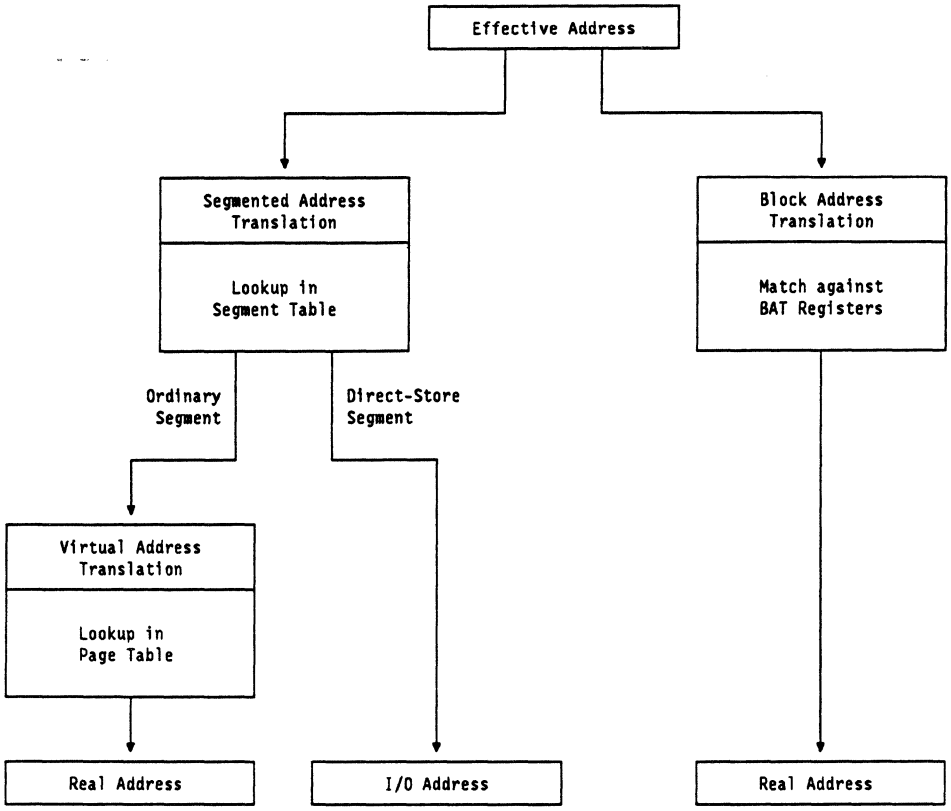


Figure 11. PowerPC Address Translation

The *Effective Address* (EA) is the address generated by the processor for load and store instructions or for instruction fetch. This address is passed simultaneously to two translation mechanisms:

- *Segmented Address Translation*, described in section 4.4 on page 23for 64-bit implementations, and in section 4.5 on page 32 for 32-bit implementations,, and
- *Block Address Translation*, described in section 4.7 on page 38.

A typical Effective Address will be successfully translated by just one of these mechanisms. If neither mechanism is successful, a *storage exception* (page 19) results.

An Effective Address that translates successfully via the Segmented Address Translation mechanism is a reference to one of two types of segments:

- A *direct-store segment*, in which case the address is converted directly into an I/O address and is passed to the I/O subsystem for further action, or
- An *ordinary segment*, in which case the address is converted into a *real address* that is then used to access storage.

An Effective Address that translates successfully via the Block Address Translation mechanism is converted directly into a *real address* that is then used to access storage.

4.4 Segmented Address Translation, 64-bit Implementations

Figure 12 shows the steps involved in translating from an Effective Address to a Real Address on a 64-bit implementation.

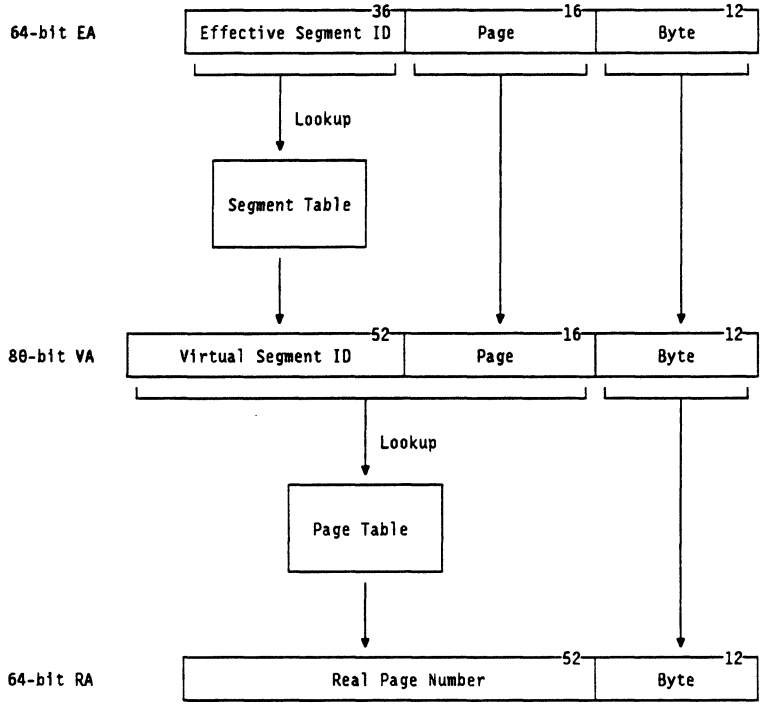


Figure 12. Address Translation Overview (64-bit implementations)

The Effective Address (EA) is a 64-bit quantity computed by the processor. Bits 0:35 of the EA are the Effective Segment ID (ESID); these are looked up in the Segment Table to produce a Virtual Segment ID (VSID). Bits 36:51 of the EA are the Page Number within the segment; these are concatenated with the VSID from the Segment Table to form the Virtual Page Number (VPN). The VPN is looked up in the Page Table to produce a Real Page Number (RPN). Bits 52:63 of the EA are the Byte Offset within the page; these are concatenated with the RPN to form the Real Address (RA) that is used to access storage.

If the processor is executing in 32-bit mode ($MSR_{SF}=0$), the translation process described above is followed except that the high-order 32 bits of the 64-bit Effective Address (that is, bits 0:31 of the ESID) are forced to zero before the lookup in the Segment Table starts. Bits 32:35 of the EA, which are the high-order 4 bits of the lower 32 bits of the EA, thus constitute the ESID.

If the selected Segment Table Entry identifies the segment as a direct-store segment, the Page Table is not referred to. Rather, translation continues as described in 4.6, "Direct-Store Segments" on page 37.

For ordinary storage segments the Segmented Address Translation mechanism may be superseded by the Block Address Translation (BAT) mechanism (see section 4.7 on page 38). If not, the translation moves in two steps from Effective Address to Virtual Address (which never exists as a specific entity but can be considered to be the concatenation of the VPN and Byte Offset), and from Virtual Address to Real Address.

The first step in segmented address translation is to convert the effective address into a virtual address, described in section 4.4.1 on page 24. The second step, conversion of the virtual address into a real address, is described in section 4.4.2 on page 28.

4.4.1 Virtual Address Generation, 64-bit Implementations

Conversion of a 64-bit Effective Address to a Virtual Address is done by searching a hashed segment table pointed to by the Address Space Register.

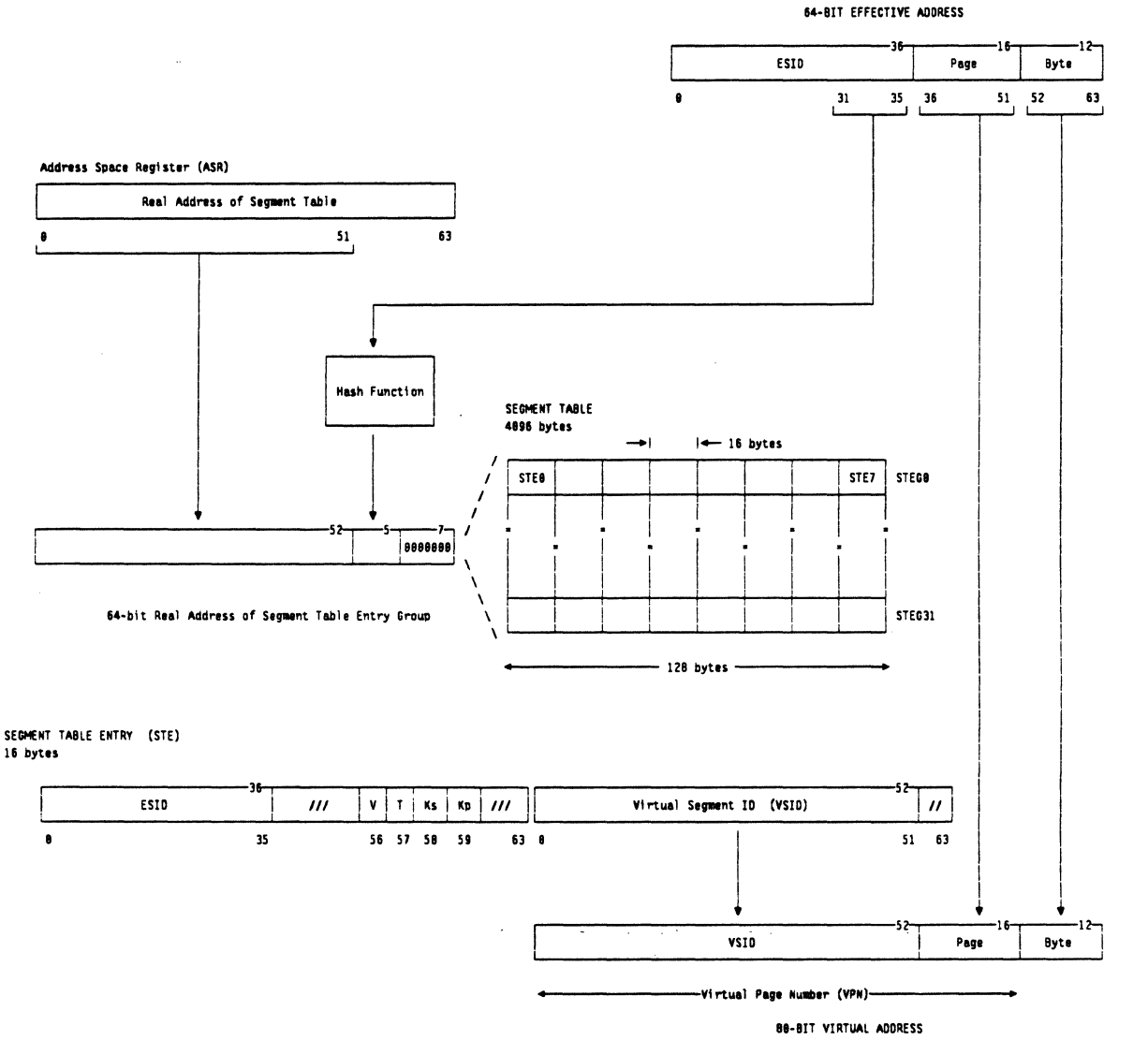


Figure 13. Translation of 64-bit Effective Address to Virtual Address

4.4.1.1 Address Space Register

The ASR is shown in Figure 14. This 64-bit special-purpose register holds the real address of the Segment Table. The Segment Table defines the set of segments than can be addressed at any one time; it is usual to have different segment tables for different processes. The contents of the ASR are usually part of the process state.

Access to the ASR is privileged. The ASR may be read or written by the ***mfspr*** and ***mtspr*** instructions. See "Move From Special Purpose Register XFX-form" on page 14 and "Move To Special Purpose Register XFX-form" on page 13.

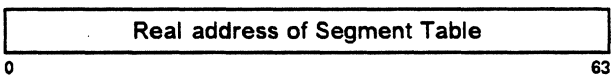
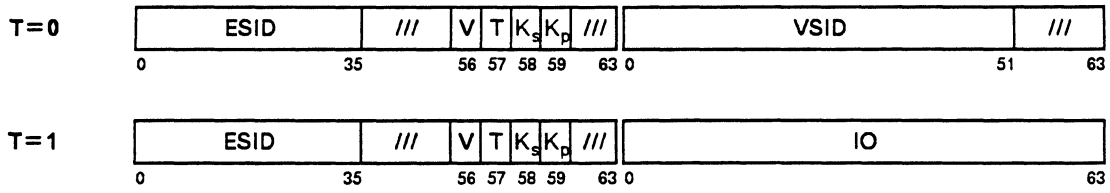


Figure 14. Address Space Register

Programming Note

The values 0, 0x1000, and 0x2000 cannot be used as Segment Table addresses, since these pages contain interrupt vectors.



Dword	*Bit*	*Name*	*Description*
0	0:35	ESID	Effective Segment ID
	56	V	Entry valid if V = 1
	57	T	Direct-store segment if T = 1
	58	K_s	Supervisor state storage key
	59	K_p	Problem state storage key
1	0:51	VSID	Virtual SID
	0:63	IO	I/O specific

All other fields are reserved.

Figure 15. Segment Table Entry format

Engineering Note

Since the Segment Table is constrained to lie on a page boundary, bits 52:63 of the ASR need not be implemented. The ***mfspr*** instruction should return a 64-bit quantity with 0's in these positions, however.

4.4.1.2 Segment Table

The Segment Table (STAB) is a one-page data structure that defines the mapping between Effective Segment IDs and Virtual Segment IDs. The STAB must be on a page boundary.

The STAB contains 32 Segment Table Entry Groups, or STEGs. A STEG contains 8 Segment Table Entries (STEs) of 16 bytes each; each STEG is thus 128 bytes long. STEGs are entry points for searches of the Segment Table.

See section 4.12, "Table Update Synchronization Requirements" on page 53 for the rules that software must follow when updating the Segment Table.

Segment Table Entry

Each Segment Table Entry (STE) maps one ESID to one VSID. Additional information in the STE controls the STAB search process and provides input to the storage protection mechanism. Figure 15 shows the layout of an STE.

See 4.10, "Storage Protection" on page 44 for a discussion of the storage key bits.

4.4.1.3 Segment Table Search

An outline of the STAB search process is shown in Figure 13 on page 24. The detailed algorithm is as follows:

1. **Primary Hash:** Bits 0:51 of the ASR are concatenated with bits 31:35 of the Effective Address (the low 5 bits of the ESID) and with a field of seven 0s to form the 64-bit real address of a Segment Table Entry Group. This operation is referred to as the "Primary STAB Hash." This identifies a particular STEG, each of whose 8 STEs will be tested in turn.
2. The first STE in the selected STEG is tested for a match with the EA. In order for a match to exist, the following must be true:
   - $STE_V = 1$
   - $STE_{ESID} = EA_{0:35}$

   If a match is found, the STE search terminates successfully.
3. Step 2 is repeated for each of the other 7 STEs in the STEG. The first matching STE terminates the search. If none of the 8 STEs match, the secondary hash must be tried.
4. **Secondary Hash:** Bits 0:51 of the ASR are concatenated with the ones-complement of bits 31:35 of the Effective Address and with a field of seven 0s to form the 64-bit real address of a Segment Table Entry Group. This operation is referred to as the "Secondary STAB Hash."
5. The first STE in the selected STEG is tested for a match with the EA. In order for a match to exist, the following must be true:
   - $STE_V = 1$
   - $STE_{ESID} = EA_{0:35}$

If a match is found, the STE search terminates successfully.

6. Step 5 is repeated for each of the other 7 STEs in the STEG. The first matching STE terminates the search. If none of the 8 STEs match, the search fails.

If the Segment Table search succeeds, the Virtual Page Number (VPN) is formed by concatenating the VSID from the matching STE with bits 36:51 of the Effective Address (the page number). The complete 80-bit Virtual Address (VA) is formed by concatenating the VPN with bits 52:63 of the EA (the byte offset).

If the search fails, a *page fault* interrupt is taken. This will be an Instruction Storage interrupt or a Data Storage interrupt, depending on whether the Effective Address is for an instruction fetch or for data access.

If the selected STE has T=1, the reference is to a direct-store segment. No reference is made to the Page Table; processing continues as described in 4.6, "Direct-Store Segments" on page 37.

Segment Lookaside Buffer

Conceptually, the segment table is searched by the address relocation hardware to translate every reference. For performance reasons the hardware usually keeps a Segment Lookaside Buffer (SLB) that holds STEs that have recently been used. The SLB is searched prior to searching the Segment Table. As a consequence, when software makes changes to the Segment Table it must perform the appropriate SLB invalidate operations to maintain the consistency of the SLB with the tables.

Programming Notes

1. Segment table entries may or may not be cached in an SLB.
2. Table lookups are done using real addresses and storage access mode M=1 (memory coherence).
3. If software plans to access the STAB with data relocate on, $MSR_{DR}=1$, it must avoid cache synonyms by mapping these tables such that the real and virtual address bits used for cache set selection are the same, just as is required for other virtual accesses. See address alignment requirements described in Book II, *PowerPC Virtual Environment Architecture*.
4. It is possible that the hardware implements two SLB arrays (one for data and one for instruction). In this case the size, shape and values contained by the arrays may be different.
5. The ASR must point to a valid Segment Table whenever address relocation is enabled ($MSR_{IR}=1$ or $MSR_{DR}=1$ or both) and the Effective Address is not covered by BAT translation.
6. Use the ***slbie***, ***slbiex***, or ***slbia*** instruction to ensure that the SLB no longer contains a mapping for a particular segment.
7. See Appendix F, "Synchronization Requirements for Special Registers" on page 83, for the synchronization requirements that must be satisfied when a program changes the contents of the ASR.
8. Hardware never modifies the Segment Table.

4.4.1.4 32-bit Execution Mode

When a 64-bit implementation executes in 32-bit mode ($MSR_{SF}=0$), the Segment Table search is modified as follows:

1. The 64-bit Effective Address is computed by the processor as usual.
2. The high-order 32 bits of the EA are forced to zero. Thus the Effective Segment ID consists of 32 0's concatenated with the high-order 4 bits of the lower half of the 64-bit EA.
3. The modified EA is then used as input to the Segment Table search.

The zeroing of the high-order 32 bits effectively truncates the 64-bit EA to a 32-bit EA such as would have been generated on a 32-bit implementation. The ESID in 32-bit mode is the high-order 4 bits of this truncated EA; the ESID thus lies in the range 0:15. These 4 bits would select a Segment Register on a 32-bit implementation; they select one of 16 STEGs in the Segment Table on a 64-bit implementation. These STEGs can be used to emulate the 32-bit machine's Segment Registers.

This truncation of the EA is the sole effect of 32-bit mode ($MSR_{SF}=0$) on address translation; everything else proceeds as for 64-bit mode.

4.4.2 Virtual to Real Translation, 64-bit Implementations

Conversion of an 80-bit Virtual Address to a Real Address is done by searching a hashed page table located by SDR 1.

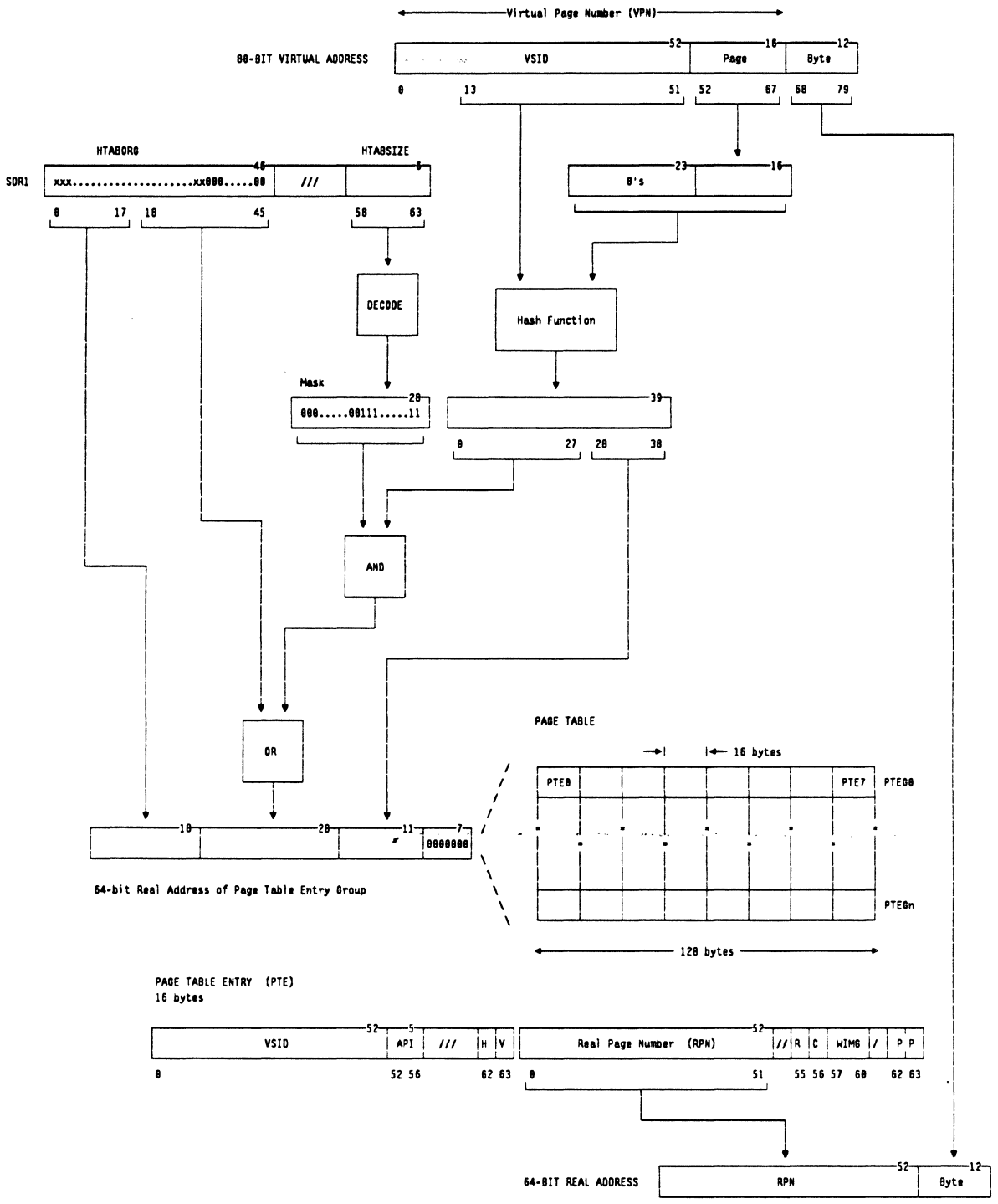


Figure 16. Translation of 80-bit Virtual Address to 64-bit Real Address

Generation of the 80-bit Virtual Address that is input to this stage of the translation process is described in 4.4.1, "Virtual Address Generation, 64-bit Implementations" on page 24.

4.4.2.1 Page Table

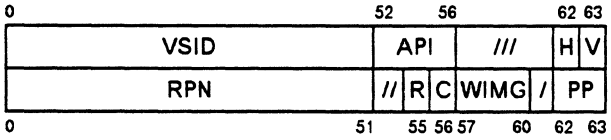
The Hashed Page Table (HTAB) is a variable-sized data structure that defines the mapping between Virtual Page Numbers and Real Page Numbers. The HTAB's size must be a power of 2, and its starting address must be a multiple of its size.

The layout of the HTAB is similar to that of the Segment Table, except that the HTAB's size is variable while the STAB's size is exactly one page. The HTAB contains a number of Page Table Entry Groups, or PTEGs. A PTEG contains 8 Page Table Entries (PTEs) of 16 bytes each; each PTEG is thus 128 bytes long. PTEGs are entry points for searches of the Page Table.

See section 4.12, "Table Update Synchronization Requirements" on page 53 for the rules that software must follow when updating the Page Table.

Page Table Entry

Each Page Table Entry (PTE) maps one VPN to one RPN. Additional information in the PTE controls the HTAB search process and provides input to the storage protection mechanism. Figure 17 shows the layout of a PTE.



Dword	*Bit*	*Name*	*Description*
0	0:51	VSID	Virtual Segment ID
	52:56	API	Abbreviated Page Index
	62	H	Hash function identifier
	63	V	Entry valid (V=1) or invalid (V=0)
1	0:51	RPN	Real Page Number
	55	R	Reference bit
	56	C	Change bit
	57:60	WIMG	Storage access controls
	62:63	PP	Page protection bits

All other fields are reserved.

Figure 17. Page Table Entry, 64-bit implementations

The PTE contains an Abbreviated Page Index rather than the complete Page field. At least 11 of the low-order bits of the VPN are used in the hash function to select a PTEG. These bits are not repeated in the PTEs of that PTEG.

Page Table Size

The number of entries in the Page Table directly affects performance because it influences the hit ratio in the Page Table and thus the rate of page fault interrupts. If the table is too small, it is possible that not all the virtual pages that actually have real page frames assigned can be mapped via the Page Table. This can happen if too many hash collisions occur and there are more than 16 entries for the same primary/secondary pair of PTEGs. While this situation cannot be guaranteed not to occur for any size Page Table, making the Page Table larger than the minimum size will reduce the frequency of occurrence of such collisions.

It is recommended that the number of PTEGs in the page table be at least one-half the number of real page frames to be mapped.

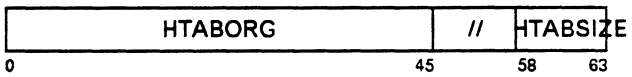
As an example, if the real memory size is 2^{31} bytes (2 GB), then we have $2^{31-12} = 2^{19}$ page frames. The minimum recommended page table size would be 2^{18} PTEGs, or 2^{25} bytes (32 MB).

Engineering Note

The minimum size page table supported on 64-bit implementations is 2048 PTEGs, or 2^{18} bytes (256 KB). This is the recommended size for a system with 2^{24} bytes (16 MB) of storage. PowerPC systems can be built with less storage, but the Page Table must be at least this minimum size.

4.4.2.2 Storage Description Register 1

The SDR 1 register is shown in Figure 18.



Bits	*Name*	*Description*
0:45	HTABORG	Real address of page table
58:63	HTABSIZE	Encoded size of table

All other fields are reserved.

Figure 18. SDR 1, 64-bit implementations

The **HTABORG** field in SDR 1 contains the high-order 46 bits of the 64-bit real address of the page table. The Page Table is thus constrained to lie on a 2^{18} byte (256 KB) boundary at a minimum. At least 11 bits from the hash function (Figure 16 on page 28) are used to index into the Page Table. The minimum size Page Table is 256 KB (2^{11} PTEGs of 128 bytes each).

The Page Table can be any size 2^n where $18 \leq n \leq 46$. As the table size is increased, more bits are used from the hash to index into the table and the value in HTABORG must have more of its low-order bits equal to 0. The **HTABSIZE** field in SDR 1 contains an integer giving the number of bits from the hash that are used in the Page Table index. HTABSIZE is used to generate a mask of the form 0b00...011...1, that is, a string of 0 bits followed by a string of 1 bits. The 1

bits determine which additional bits (beyond the minimum of 11) from the hash are used in the index; HTABORG must have this same number of low-order bits equal to 0. See Figure 16 on page 28.

Engineering Note

The number of low-order 0 bits in HTABORG must be at least the value in HTABSIZE so that the final 64-bit real address can be formed by ORing the various components.

Example

Suppose that the Page Table is 16,384 (2^{14}) 128-byte PTEGs, for a total size of 2^{21} bytes (2 MB). A 14-bit index is required. Eleven bits are provided from the hash to start with, so 3 additional bits from the hash must be selected. Thus the value in HTABSIZE must be 3 and the value in HTABORG must have its low-order 3 bits (bits 31:33 of SDR 1) equal to 0. This means that the Page Table must begin on a $2^{3+11+7} = 2^{21} = 2$ MB boundary.

4.4.2.3 Hashed Page Table Search

An outline of the HTAB search process is shown in Figure 16 on page 28. The detailed algorithm is as follows:

1. **Primary Hash:** A 39-bit hash value is computed by Exclusive-ORing the low-order 39 bits of the VSID with a 39-bit value formed by concatenating 23 bits of 0 with the Page index.
2. The 64-bit real address of a PTEG is formed by concatenating the following values:
   - Bits 0:17 of SDR 1 (the 18 high-order bits of HTABORG).
   - Bits 0:27 of the value formed in step 1 ANDed with the mask generated from bits 58:63 of SDR 1 (HTABSIZE) and then ORed with bits 18:45 of SDR 1 (the 28 low-order bits of HTABORG).
   - Bits 28:38 of the value formed in step 1.
   - A 7-bit field of 0s.

   This operation is referred to as the "Primary HTAB Hash." This identifies a particular PTEG, each of whose 8 PTEs will be tested in turn.
3. The first PTE in the selected PTEG is tested for a match with VPN. In order for a match to exist, the following must be true:
   - $PTE_H = 0$
   - $PTE_V = 1$
   - $PTE_{VSID} = VA_{0:51}$
   - $PTE_{API} = VA_{52:56}$

   If a match is found, the PTE search terminates successfully.
4. Step 3 is repeated for each of the other 7 PTEs in the PTEG. The first matching PTE terminates the search. If none of the 8 PTEs match, the secondary hash must be tried.
5. **Secondary Hash:** A 39-bit hash value is computed by taking the ones complement of the Exclusive OR of the low-order 39 bits of the VSID with a 39-bit value formed by concatenating 23 bits of 0 with the Page index.
6. The 64-bit real address of a PTEG is formed by concatenating the following values:
   - Bits 0:17 of SDR 1 (the 18 high-order bits of HTABORG).
   - Bits 0:27 of the value formed in step 5 ANDed with the mask generated from bits 58:63 of SDR 1 (HTABSIZE) and then ORed with bits 18:45 of SDR 1 (the 28 low-order bits of HTABORG).
   - Bits 28:38 of the value formed in step 5.
   - A 7-bit field of 0s.

   This operation is referred to as the "Secondary HTAB Hash."
7. The first PTE in the selected PTEG is tested for a match with VPN. In order for a match to exist, the following must be true:
   - $PTE_H = 1$
   - $PTE_V = 1$
   - $PTE_{VSID} = VA_{0:51}$
   - $PTE_{API} = VA_{52:56}$

   If a match is found, the PTE search terminates successfully.
8. Step 7 is repeated for each of the other 7 PTEs in the PTEG. The first matching PTE terminates the search. If none of the 8 PTEs match, the search fails.

If the Page Table search succeeds, the content of the PTE that translates the EA is returned. The Real Address (RA) is formed by concatenating the RPN from the matching PTE with bits 52:63 of the Effective Address (the byte offset).

If the search fails, a *page fault* interrupt is taken. This will be an Instruction Storage interrupt or a Data Storage interrupt, depending on whether the Effective Address is for an instruction fetch or for data access.

Translation Lookaside Buffer

Conceptually, the Page Table is searched by the address relocation hardware to translate every reference. For performance reasons the hardware usually keeps a Translation Lookaside Buffer (TLB) that holds PTEs that have recently been used. The TLB is searched prior to searching the Page Table. As a consequence, when software makes changes to the Page Table it must perform the appropriate TLB invalidate operations to maintain the consistency of the TLB with the Page Table.

Programming Notes

1. Page table entries may or may not be cached in a TLB.
2. Table lookups are done using real addresses and storage access mode M=1 (memory coherence).
3. If software plans to access the HTAB with data relocate on, $MSR_{DR}=1$, it must avoid cache synonyms by mapping these tables such that the real and virtual address bits used for cache set selection are the same, just as is required for other virtual accesses. See address alignment requirements described in Book II, *PowerPC Virtual Environment Architecture*.
4. It is possible that the hardware implements two TLB arrays (one for data and one for instruction). In this case the size, shape and values contained by the arrays may be different.
5. Use the ***tlbie***, ***tlbiex***, or ***tlbia*** instruction to ensure that the TLB no longer contains a mapping for a particular page.
6. Refer to Book IV, *PowerPC Implementation Features* for the procedure to be used to invalidate the entire TLB.

4.5 Segmented Address Translation, 32-bit Implementations

Figure 19 shows the steps involved in translating from an effective address to a real address on a 32-bit implementation.

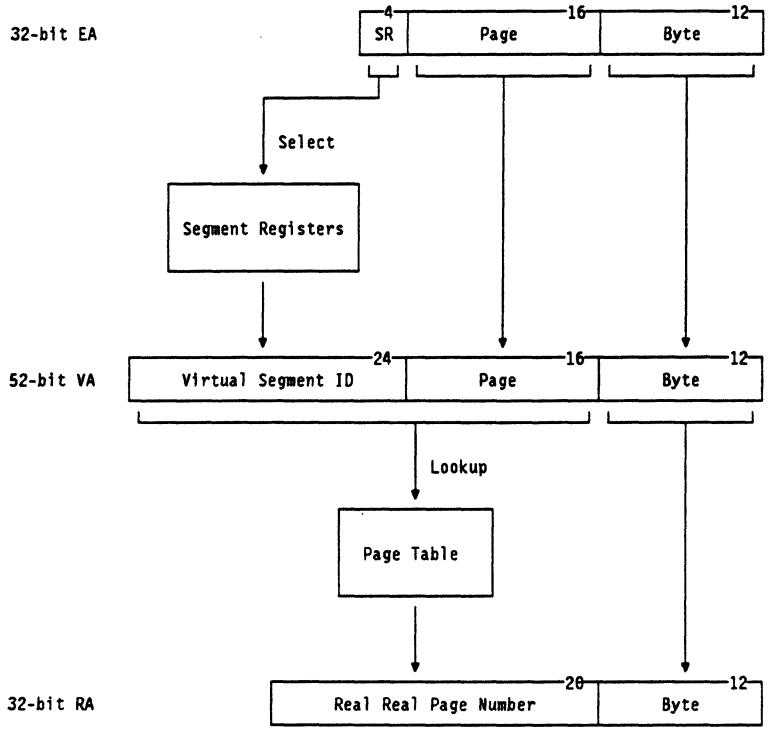


Figure 19. Address Translation Overview (32-bit implementations)

The Effective Address (EA) is a 32-bit quantity computed by the processor. Bits 0:3 of the EA are the Segment Register number. These are used to select a Segment Register, from which is extracted a Virtual Segment ID (VSID). Bits 4:19 of the EA are the Page Number within the segment; these are concatenated with the VSID from the Segment Register to form the Virtual Page Number (VPN). The VPN is looked up in the Page Table to produce a Real Page Number (RPN). Bits 20:31 of the EA are the Byte Offset within the page; these are concatenated with the RPN to form the Real Address (RA) that is used to access storage.

If the selected Segment Register identifies the segment as a direct-store segment, the Page Table is not referred to. Rather, translation continues as described in 4.6, "Direct-Store Segments" on page 37.

For ordinary storage segments the Segmented Address Translation mechanism may be superseded by the Block Address Translation (BAT) mechanism (see section 4.7 on page 38). If not, the translation moves in two steps from Effective Address to Virtual Address (which never exists as a specific entity but can be considered to be the concatenation of the VPN and Byte Offset), and from Virtual Address to Real Address.

The first step in segmented address translation is to convert the effective address into a virtual address, described in section 4.5.1 on page 33. The second step, conversion of the virtual address into a real address, is described in section 4.5.2 on page 34.

4.5.1 Virtual Address Generation, 32-bit Implementations

Conversion of a 32-bit Effective Address to a Virtual Address is done by using the 4 high-order bits of the EA to select a Segment Register.

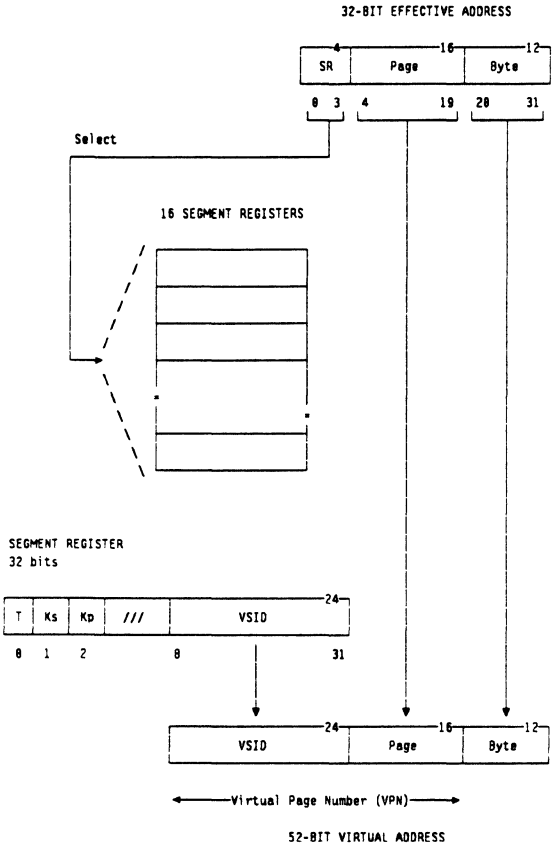
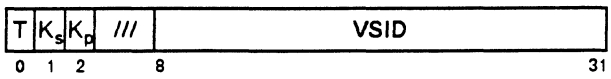


Figure 20. Translation of 32-bit Effective Address to Virtual Address

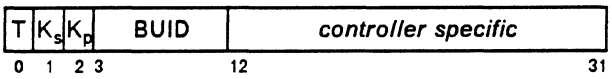
4.5.1.1 Segment Registers

The 16 32-bit registers are present only in 32-bit implementations of PowerPC. Figure 21 shows the layout of a Segment Register. The fields in the Segment Register are interpreted differently depending on the value of bit 0 (the T bit).



Bit	*Name*	*Description*
0	T	T = 0 selects this format
1	K_s	Supervisor state storage key
2	K_p	Problem state storage key
8:31	VSID	Virtual Segment ID

All other fields are reserved



Bit	*Name*	*Description*
0	T	T = 1 selects this format
1	K_s	Supervisor state storage key
2	K_p	Problem state storage key
3:11	BUID	Bus Unit ID
12:31		Device dependent data for I/O controller

Figure 21. Segment Register format

If T = 0 in the selected Segment Register, the Effective Address is a reference to an ordinary storage segment. For ordinary segments the Segmented Address Translation mechanism may be superseded by the Block Address Translation (BAT) mechanism (see section 4.7 on page 38). If not, the 52-bit Virtual Address (VA) is formed by concatenating

- the 24-bit VSID field from the Segment Register.
- the 16-bit page index, $EA_{4:19}$, and
- the 12-bit byte offset, $EA_{20:31}$.

The VA is then translated to a Real Address as described in the next section.

If T = 1 in the selected Segment Register, the Effective Address is a reference to a direct-store segment. No reference is made to the page table; processing continues as in 4.6, "Direct-Store Segments" on page 37.

4.5.2 Virtual to Real Translation, 32-bit Implementations

Conversion of a 52-bit Virtual Address to a Real Address is done by searching a hashed page table located by SDR 1.

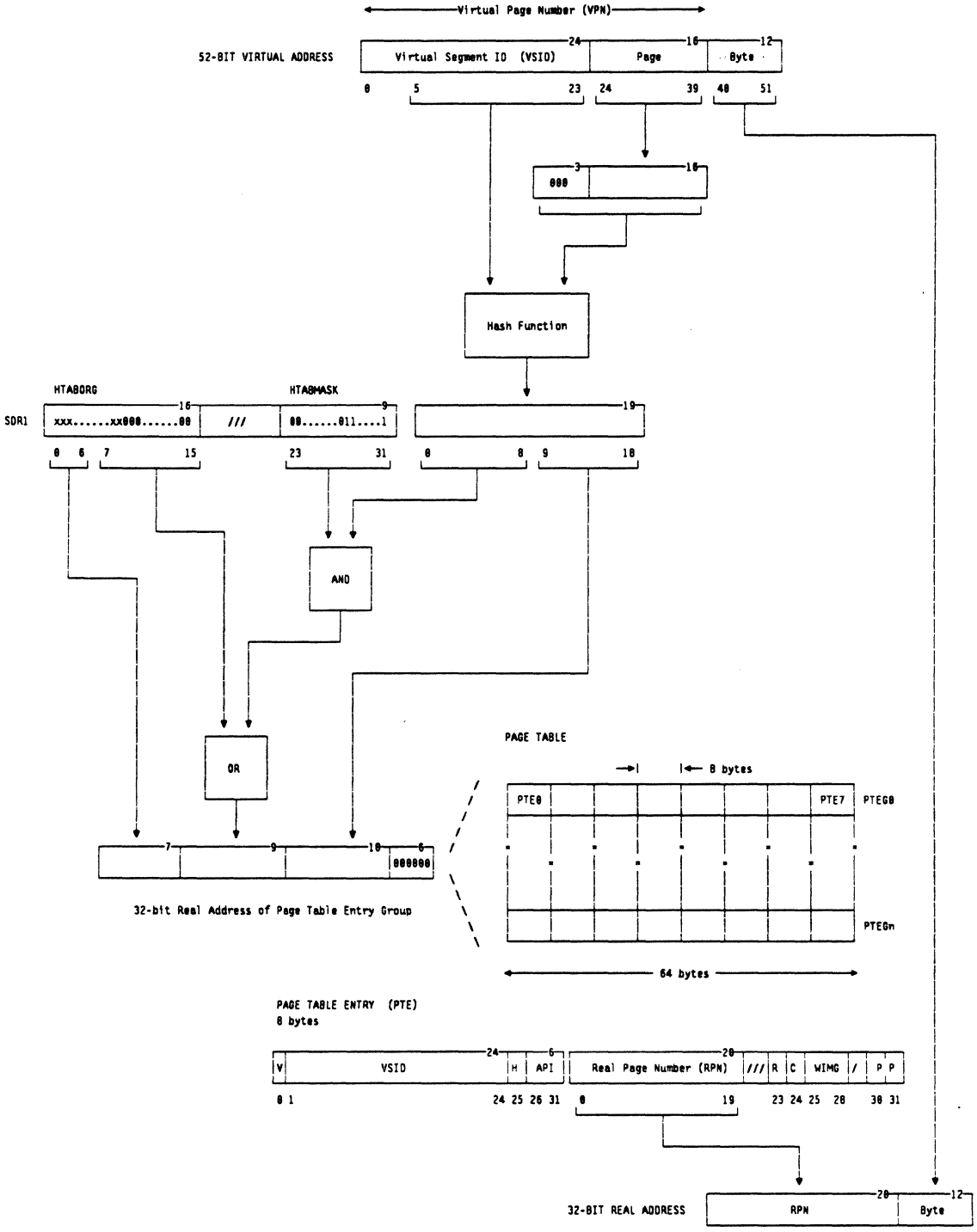


Figure 22. Translation of 52-bit Virtual Address to 32-bit Real Address

Generation of the 52-bit Virtual Address that is input to this stage of the translation process is described in 4.5.1, "Virtual Address Generation, 32-bit Implementations" on page 33.

4.5.2.1 Page Table

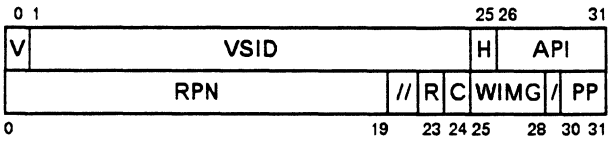
The Hashed Page Table (HTAB) is a variable-sized data structure that defines the mapping between Virtual Page Numbers and Real Page Numbers. The HTAB's size must be a power of 2, and its starting address must be a multiple of its size.

The HTAB contains a number of Page Table Entry Groups, or PTEGs. A PTEG contains 8 Page Table Entries (PTEs) of 8 bytes each; each PTEG is thus 64 bytes long. PTEGs are entry points for searches of the Page Table.

See section 4.12, "Table Update Synchronization Requirements" on page 53 for the rules that software must follow when updating the Page Table.

Page Table Entry

Each Page Table Entry (PTE) maps one VPN to one RPN. Additional information in the PTE controls the HTAB search process and provides input to the storage protection mechanism. Figure 23 shows the layout of a PTE.



Word	*Bit*	*Name*	*Description*
0	0	V	Entry valid (V=1) or invalid (V=0)
	1:24	VSID	Virtual Segment ID
	25	H	Hash function identifier
	26:31	API	Abbreviated Page Index
1	0:19	RPN	Real Page Number
	23	R	Reference bit
	24	C	Change bit
	25:28	WIMG	Storage access controls
	30:31	PP	Page protection bits

All other fields are reserved.

Figure 23. Page Table Entry, 32-bit implementations

The PTE contains an Abbreviated Page Index rather than the complete Page field. At least 10 of the low-order bits of the Page are used in the hash function to select a PTEG. These bits are not repeated in the PTEs of that PTEG.

Page Table Size

The number of entries in the Page Table directly affects performance because it influences the hit ratio in the Page Table and thus the rate of page fault interrupts. If the table is too small, it is possible that not all the virtual pages that actually have real page frames assigned can be mapped via the Page Table. This can happen if too many hash collisions occur and there are more than 16 entries for the same primary/secondary pair of PTEGs. While this situation cannot be guaranteed not to occur for any size Page Table, making the Page Table larger than the minimum size will reduce the frequency of occurrence of such collisions.

It is recommended that the number of PTEGs in the page table be at least one-half the number of real page frames to be mapped.

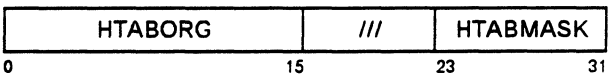
As an example, if the real memory size is 2^{29} bytes (512 MB), then we have $2^{29-12} = 2^{17}$ page frames. The minimum recommended page table size would be 2^{16} PTEGs, or 2^{22} bytes (4 MB).

Engineering Note

The minimum size page table supported on 32-bit implementations is 1024 PTEGs, or 2^{16} bytes (64 KB). This is the recommended size for a system with 2^{23} bytes (8 MB) of storage. PowerPC systems can be built with less storage, but the Page Table must be at least this minimum size.

4.5.2.2 Storage Description Register 1

The SDR 1 register is shown in Figure 24.



Bits	*Name*	*Description*
0:15	HTABORG	Real address of page table
23:31	HTABMASK	Mask for page table address

All other fields are reserved.

Figure 24. SDR 1, 32-bit implementations

Architecture Note

In SDR 1 on 64-bit implementations, the HTABSIZE field contains a number that specifies the number of 1 bits in the Page Table index mask. On 32-bit implementations the mask itself is contained in the HTABMASK field.

The **HTABORG** field in SDR 1 contains the high-order 16 bits of the 32-bit real address of the page table. The Page Table is thus constrained to lie on a 2^{16} byte (64 KB) boundary at a minimum. At least 10 bits from the hash function (Figure 22 on page 34) are used to index into the Page Table. The minimum size Page Table is 64 KB (2^{10} PTEGs of 64 bytes each).

The Page Table can be any size 2^n where $16 \leq n \leq 25$. As the table size is increased, more bits are used from the hash to index into the table and the value in

HTABORG must have more of its low-order bits equal to 0. The **HTABMASK** field in SDR 1 contains a mask value that determines how many bits from the hash are used in the Page Table index. This mask must be of the form 0b00...011...1, that is, a string of 0 bits followed by a string of 1 bits. The 1 bits determine how many additional bits (beyond the minimum of 10) from the hash are used in the index; HTABORG must have this same number of low-order bits equal to 0. See Figure 22 on page 34.

Engineering Note

The number of low-order 0 bits in HTABORG must be at least the number of 1 bits in HTABMASK so that the final 32-bit real address can be formed by ORing the various components.

Example

Suppose that the Page Table is 8,192 (2^{13}) 64-byte PTEGs, for a total size of 2^{19} bytes (512 KB). A 13-bit index is required. Ten bits are provided from the hash to start with, so 3 additional bits from the hash must be selected. Thus the value in HTABMASK must be 0x007 and the value in HTABORG must have its low-order 3 bits (bits 13:15 of SDR 1) equal to 0. This means that the Page Table must begin on a $2^{3+10+6} = 2^{19} = 512$ KB boundary.

4.5.2.3 Hashed Page Table Search

An outline of the HTAB search process is shown in Figure 22 on page 34. The detailed algorithm is as follows:

1. A 19-bit hash value is computed by Exclusive-ORing the low-order 19 bits of the VSID with a 19-bit value formed by concatenating 3 bits of 0 with the Page index.
2. **Primary Hash:** The 32-bit real address of a PTEG is formed by concatenating the following values:
   - Bits 0:6 of SDR 1 (the 7 high-order bits of HTABORG).
   - Bits 0:8 of the value formed in step 1 ANDed with bits 23:31 of SDR 1 (the value of HTABMASK) and then ORed with bits 7:15 of SDR1 (the 9 low-order bits of HTABORG).
   - Bits 9:18 of the value formed in step 1.
   - A 6-bit field of 0s.

   This operation is referred to as the "Primary HTAB Hash." This identifies a particular PTEG, each of whose 8 PTEs will be tested in turn.
3. The first PTE in the selected PTEG is tested for a match with VPN. In order for a match to exist, the following must be true:
   - $PTE_H = 0$
   - $PTE_V = 1$
   - $PTE_{VSID} = VA_{0:23}$
   - $PTE_{API} = VA_{24:29}$

   If a match is found, the PTE search terminates successfully.
4. Step 3 is repeated for each of the other 7 PTEs in the PTEG. The first matching PTE terminates the search. If none of the 8 PTEs match, the secondary hash must be tried.
5. A 19-bit hash value is computed by taking the ones complement of the Exclusive OR of the low-order 19 bits of the VSID with a 19-bit value formed by concatenating 3 bits of 0 with the Page index.
6. **Secondary Hash:** The 32-bit real address of a PTEG is formed by concatenating the following values:
   - Bits 0:6 of SDR 1 (the 7 high-order bits of HTABORG).
   - Bits 0:8 of the value formed in step 5 ANDed with bits 23:31 of SDR 1 (the value of HTABMASK) and then ORed with bits 7:15 of SDR1 (the 9 low-order bits of HTABORG).
   - Bits 9:18 of the value formed in step 5.
   - A 6-bit field of 0s.

   This operation is referred to as the "Secondary HTAB Hash."
7. The first PTE in the selected PTEG is tested for a match with VPN. In order for a match to exist, the following must be true:
   - $PTE_H = 1$
   - $PTE_V = 1$
   - $PTE_{VSID} = VA_{0:23}$
   - $PTE_{API} = VA_{24:29}$

   If a match is found, the PTE search terminates successfully.
8. Step 7 is repeated for each of the other 7 PTEs in the PTEG. The first matching PTE terminates the search. If none of the 8 PTEs match, the search fails.

If the Page Table search succeeds, the content of the PTE that translates the EA is returned. The Real Address (RA) is formed by concatenating the RPN from the matching PTE with bits 20:31 of the Effective Address (the byte offset).

If the search fails, a *page fault* interrupt is taken. This will be an Instruction Storage interrupt or a Data Storage interrupt, depending on whether the Effective Address is for an instruction fetch or for data access.

Translation Lookaside Buffer

Conceptually, the Page Table is searched by the address relocation hardware to translate every reference. For performance reasons the hardware usually keeps a Translation Lookaside Buffer (TLB) that holds PTEs that have recently been used. The TLB is searched prior to searching the Page Table. As a consequence, when software makes changes to the

Page Table it must perform the appropriate TLB invalidate operations to maintain the consistency of the TLB with the Page Table.

Programming Notes

1. Page table entries may or may not be cached in a TLB.
2. Table lookups are done using real addresses and storage access mode M=1 (memory coherence).
3. If software plans to access the HTAB with data relocate on, $MSR_{DR}=1$, it must avoid cache synonyms by mapping these tables such that the real and virtual address bits used for cache set selection are the same, just as is required for other virtual accesses. See address alignment requirements described in Book II, *PowerPC Virtual Environment Architecture*.
4. It is possible that the hardware implements two TLB arrays (one for data and one for instruction). In this case the size, shape and values contained by the arrays may be different.
5. Use the ***tlbie***, ***tlbiex***, or ***tlbia*** instruction to ensure that the TLB no longer contains a mapping for a particular page.
6. Refer to Book IV, *PowerPC Implementation Features* for the procedure to be used to invalidate the entire TLB.

4.6 Direct-Store Segments

A *direct-store* segment is a mapping of effective addresses onto an external address space, typically an I/O bus.

Compatibility Note

Direct-store segments are provided for Power compatibility. Applications that require low-latency load/store access to external address space should consider more traditional methods.

Effective addresses that lie within direct-store segments complete only the first step of the ordinary segmented address translation.

- In *64-bit implementations*, this is the search of the Segment Table. If the resulting Segment Table Entry has T=1, the reference is to a direct-store segment.
- In *32-bit implementations*, this is the selection of the Segment Register. If the SR has T=1, the reference is to a direct-store segment.

4.6.1 Completion of direct-store access

Once the segmented address translation process has discovered that the segment has T=1, translation terminates. Any match due to Block Address Translation (BAT, section 4.7) is ignored. No reference is made to the Page Table; reference and change bits are not updated. The following data is sent to the storage controller:

For 64-bit implementations:

- A one bit field representing the privilege of the storage access, computed as follows:

  $Key \leftarrow (K_p \ \& \ MSR_{PR}) \ | \ (K_s \ \& \ \neg MSR_{PR})$

- The 32-bit IO field from bits 32:63 of the second doubleword of the STE
- The low-order 28 bits of the Effective Address, $EA_{36:63}$

For 32-bit implementations:

- A one bit field representing the privilege of the storage access, computed as follows:

  $Key \leftarrow (K_p \ \& \ MSR_{PR}) \ | \ (K_s \ \& \ \neg MSR_{PR})$

- The contents of bits 3:31 of the Segment Register, which is the BUID field concatenated with the "controller specific" field.
- The low-order 28 bits of the Effective Address, $EA_{4:31}$

An implementation of PowerPC Architecture may cause multiple address/data transfers for a single instruction. The address for each transfer will be

handled in the same manner that addresses for access to main store are handled.

Architecture Note

PowerPC differs from Power in this area. Power implementations sent the address and byte count to the storage controller, causing only one address transfer regardless of the number of bytes transferred.

4.6.2 Direct-store segment protection

Page-level protection as described in 4.10.1, "Page Protection" on page 44 is not provided by the PowerPC processor for direct-store segments. The appropriate key bit (K_s or K_p) from the STE or SR is sent to the storage controller, but it is up to the storage controller to implement any protection mechanism. Frequently no such mechanism will be provided; the fact that a direct-store segment is mapped into the address space of a process may be regarded as sufficient authority to access the segment.

4.6.3 Instructions not supported for T=1

The following instructions are not supported when issued with an Effective Address in a segment where T=1:

- ***lwarx***
- ***ldarx***
- ***eciwx***
- ***stwcx.***
- ***stdcx.***
- ***ecowx***

If one of these instructions is executed with an effective address in a segment with T=1, a Data Storage interrupt may occur or the results may be boundedly undefined.

4.6.4 Instructions with no effect for T=1

The following instructions are treated as no-ops when issued with an Effective Address in a segment where T=1:

- ***dcbt***
- ***dcbtst***
- ***dcbf***
- ***dcbi***
- ***dcbst***
- ***dcbz***
- ***icbi***

For further details of storage references to direct-store segments, refer to Book IV, *PowerPC Implementation Features*.

4.7 Block Address Translation

The *Block Address Translation* (BAT) mechanism provides a means for mapping ranges of virtual addresses larger than a single page onto contiguous areas of real storage. Such areas can be used for data that is not subject to normal virtual storage handling (paging), such as a memory-mapped display buffer or an extremely large array of numerical data.

4.7.1 Recognition of Addresses in BAT Areas

Block Address Translation is enabled only when address translation is enabled ($MSR_{IR}=1$ or $MSR_{DR}=1$ or both) and then only for segments that specify T=0. That is, BAT *does not apply* to direct-store (T=1) segments.

A set of Special Purpose Registers (SPRs) called BAT registers define the starting addresses and sizes of BAT areas. The BAT registers are accessed in parallel with segmented address translation to determine whether a particular EA corresponds to a BAT area. If an EA is within a BAT area, the real address for storage access is determined as described below.

It is possible to set up the BAT registers and the segmented address translation mechanism such that a particular Effective Address is within a BAT area and also is covered by page translation. When this happens, the translation that is used is determined as follows:

MSR_{DR}, MSR_{IR}	STE or Segment Reg "T" bit	Address Translation
0	-	None (real addressing)
1	0	BAT prevails
1	1	Segment prevails

Programming Note

It is possible for a BAT area to overlay part of an ordinary segment, such that the BAT portion is non-pagable while the rest of the segment is pageable. If this is done, it is not necessary to supply Page Table entries for the portion of the segment overlaid by the BAT.

The BAT areas are defined by pairs of SPRs. These SPRs can be read or written by the ***mfspr*** and ***mtspr*** instructions; see page 14. Access to these SPRs is privileged. The layout of the BAT registers is shown in figure 25 for 64-bit implementations and in figure 26 for 32-bit implementations.

Four pairs of BAT registers are provided for translating instruction addresses (the IBAT registers), and four pairs are provided for translating data addresses (the DBAT registers).

Programming Note

If the same storage address is to be mapped via BAT for both I-fetch and data load and store, it is necessary to load the mapping into both an IBAT pair and a DBAT pair. This is true even on an implementation that does not have split I and D caches.

It is an error for system software to set up the BAT registers such that an Effective Address is translated by more than one IBAT pair or by more than one DBAT pair. If this occurs, the results are undefined and may include a violation of the storage protection mechanism, a Machine Check interrupt, or a Checkstop.

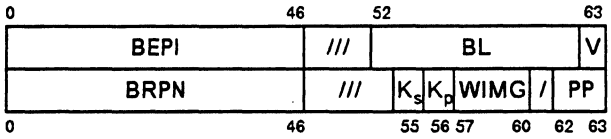
Each pair of BAT registers defines the starting address of a BAT area in Effective Address space, the length of the area, and the start of the corresponding area in Real Address space. If an Effective Address is within the range of EAs defined by a pair of BAT registers, its Real Address is developed by (conceptually) subtracting the starting effective address of the BAT area from the EA and adding the starting real address of the BAT area.

BAT areas are restricted to a finite set of allowable lengths, all of which are powers of 2. The smallest BAT area defined is 128 KB (2^{17} bytes). The largest BAT area defined is 256 MB (2^{28} bytes). The starting address of a BAT area in both EA space and RA space must be a multiple of the area's length.

4.7.2 BAT Registers

See section "Move To Special Purpose Register XFX-form" on page 13 for a list of the SPR numbers for the BAT registers. See Appendix B, "Assembler Extended Mnemonics" on page 75 for a list of extended mnemonics for use with the BAT registers.

Upper BAT Register



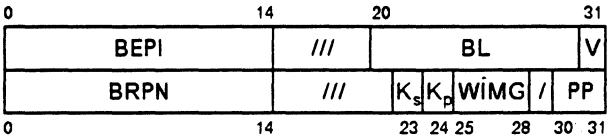
Lower BAT Register

Reg	*Bit*	*Name*	*Description*
Upper	0:46	BEPI	Block Effective Page Index
	52:62	BL	Block Length
	63	V	BAT pair valid if V=1
Lower	0:46	BRPN	Block Real Page Number
	55	K_s	Supervisor state storage key
	56	K_p	Problem state storage key
	57:60	WIMG	Storage access controls
	62:63	PP	Protection bits for BAT area

All other fields are reserved.

Figure 25. BAT Registers, 64-bit implementations

Upper BAT Register



Lower BAT Register

Reg	*Bit*	*Name*	*Description*
Upper	0:14	BEPI	Block Effective Page Index
	20:30	BL	Block Length
	31	V	BAT pair valid if V=1
Lower	0:14	BRPN	Block Real Page Number
	23	K_s	Supervisor state storage key
	24	K_p	Problem state storage key
	25:28	WIMG	Storage access controls
	30:31	PP	Protection bits for BAT area

All other fields are reserved.

Figure 26. BAT Registers, 32-bit implementations

The BL field in the lower BAT register is a mask that encodes the length of the BAT area.

BAT Area Length	*BL*
128 KB	000 0000 0000
256 KB	000 0000 0001
512 KB	000 0000 0011
1 MB	000 0000 0111
2 MB	000 0000 1111
4 MB	000 0001 1111
8 MB	000 0011 1111
16 MB	000 0111 1111
32 MB	000 1111 1111
64 MB	001 1111 1111
128 MB	011 1111 1111
256 MB	111 1111 1111

Only the values shown are valid for BL. The rightmost bit of BL is aligned with bit 46 {14} of the EA.

An Effective Address is determined to be within a BAT area if EA matches BEPI. The boundary between the string of 0s and the string of 1s in BL determines the bits of EA that participate in the comparison with BEPI: bits in EA corresponding to 1s in BL are forced to 0 for this comparison.

Bits in EA corresponding to 1s in BL, concatenated with the 17 bits of EA to the right of BL, form the offset within the BAT area.

Programming Note

The value loaded into BL determines both the length of the BAT area and the alignment of the area in both EA space and RA space. It is a programming error if the value loaded into BL is not one of those given in the table above, or if the values loaded into BEPI and BRPN do not have at least as many low-order 0s as there are 1s in BL.

4.7.2.1 BAT Storage Protection

If an Effective Address is determined to be within a BAT area, the access is next validated by the storage protection scheme described in section 4.10.2, "BAT Protection" on page 44. If this protection mechanism rejects the EA, a page fault (Data Storage interrupt or Instruction Storage interrupt) is generated.

4.7.2.2 BAT Real Address

If the protection mechanism accepts the access, then a Real Address is formed as shown in figure 27 for 64-bit implementations, and figure 28 for 32-bit implementations.

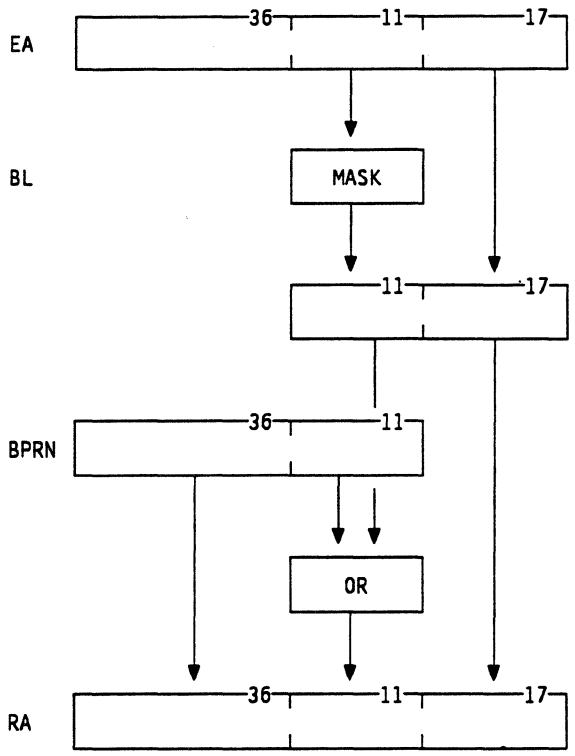


Figure 27. Formation of Real Address via BAT, 64-bit implementations

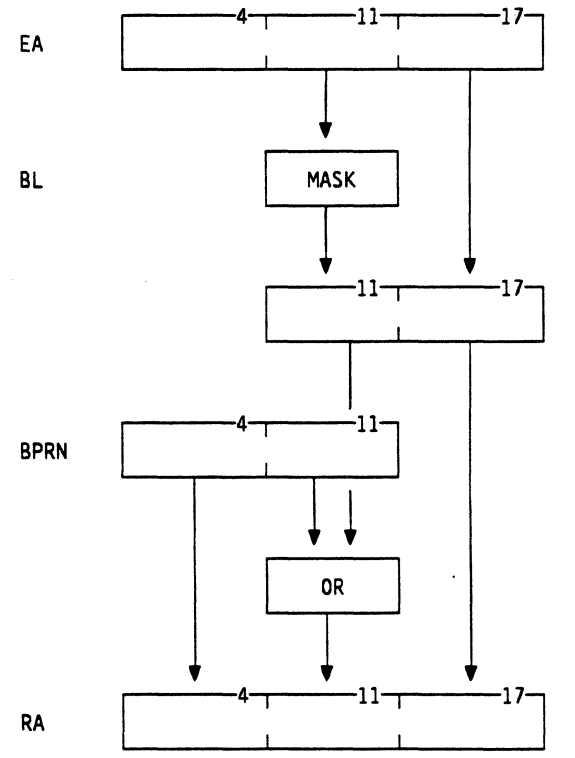


Figure 28. Formation of Real Address via BAT, 32-bit implementations.

Access to the real memory of the BAT area is made according to the storage mode defined by the "WIMG" bits in the lower BAT register. These bits apply to the

entire BAT area rather than to an individual page. See 4.8.2, "Supported Storage Modes" on page 42 for an explanation of these bits.

4.8 Storage Access Modes

When address relocation is enabled and the effective address generated by a storage access is translated by the Segmented Address Translation mechanism or by the Block Address Translation mechanism, the access is performed under the control of the Page Table Entry or BAT entry used to translate the effective address. Each entry contains four mode control bits, ***W***, ***I***, ***M***, and **G**, that specify the storage mode for all accesses translated by the entry. The ***W*** and ***I*** bits control how the processor executing the access uses its own cache. The ***M*** bit specifies whether the processor executing the access must use the storage coherence protocol to ensure that all copies of the addressed storage location are made consistent. The **G** bit controls whether or not speculative data and instruction fetching is permitted.

The mode control bits only have meaning when an effective address is translated in the processor performing a storage access. When an access is performed for which coherence is required, the processor performing the access must inform the coherence mechanism that the access requires memory coherence. Other processors affected by the access must respond to the coherence mechanism. However since these mode control bits are only relevant when an effective address is translated and have no direct relation to data in the cache, processors responding to the coherence request are able to respond without knowledge of the state of these bits.

4.8.1 W, I, M and G bits

The ***W***, ***I***, ***M***, and **G** bits in a Page Table Entry or in a BAT register control the way in which the processor accesses cache and main storage. Each bit controls a separate aspect of storage references.

W *Write Through*

If the data is in the cache, a store must update that copy of the data. In addition, if W = 1 the update must be written to the home storage location (see below).

Store combining optimizations are allowed except when the store instructions are separated by ***sync*** or ***eieio***. The architecture presumes that data present in the cache is valid and a store may cause any part of that data to be copied back to main storage.

The definition of the home storage location is dependent upon the implementation of the memory system but can be illustrated by the following examples:

- RAM Storage
  The store must be sent to the RAM controller to be written into the target RAM.
- I/O Adapter Card
  the store must be sent to the adapter card to be written to the target register or storage location.

In systems with multilevel caching, the store must be written to at least a depth in the memory hierarchy that is seen by all processors and devices.

I *Caching Inhibited*

If I = 1, the storage access is completed by referencing the location in main storage, bypassing the cache. During the access, the accessed location is not brought into the cache nor is the location allocated in the cache. It is considered a programming error if a copy of the target location of an access to Caching Inhibited storage is in the cache. Software must ensure that the location has not previously been brought into the cache or, if it has, that it has been flushed from the cache. If the programming error occurs, the result of the access is boundedly undefined.

Load/store combining optimizations are allowed except when the accesses are separated by ***sync*** or ***eieio***.

M *Memory Coherence*

This mode control is provided to allow improved performance in systems in which accesses to storage kept consistent by hardware is slower than accesses to storage not kept consistent by hardware, and in which software is able to enforce the required consistency. When the mode is off (M = 0), the hardware need not enforce data coherence. When the mode is on (M = 1), the hardware must enforce data coherence.

System Note

Entities other than processors can request that their memory transactions obey memory coherence.

Engineering Note

Since instruction storage need not be consistent with data storage, instruction fetches may be originated as noncoherent requests, regardless of the page's M bit. This can result in better performance in an implementation in which a coherent storage request has greater latency or overhead than a noncoherent storage request.

G *Guarded Storage*

This storage attribute is independent of the other three attributes. The processor will not speculatively access storage for which G=1 whether for instruction fetch or data access, except that if an instruction will be executed, the entire cache block containing that instruction may be loaded, and if a load or store operation will be executed, the entire cache block(s) containing the referenced data may be loaded into the cache.

4.8.2 Supported Storage Modes

The combinations of the write through bit, the caching inhibited bit, and the memory coherence bit define eight different storage modes. Six of these modes are supported. For each, the G bit may be 0 or 1.

- **WIM = 000**
  1. Data may be cached.
  2. Loads or stores for which the target location is in the cache may use that copy of the location.
  3. Exclusive ownership of the block containing the target location is not required for store accesses and consistency operations for the block may be ignored when fetching the block, storing it back, or changing its state from shared to exclusive.
- **WIM = 001**
  1. Data may be cached.
  2. Loads or stores for which the target location is in the cache may use that copy of the location.
  3. Exclusive ownership of the block containing the target location is required before store accesses are allowed. When fetching the block, the processor must indicate that consistency is to be enforced on the bus transaction. If the state of the block is read shared, the processor must gain exclusive use of the block before storing into it.
- **WIM = 010**

  Caching is inhibited. The storage access goes to storage bypassing the cache. Hardware enforced storage consistency is not required.
- **WIM = 011**

  Caching is inhibited. The storage access goes to storage bypassing the cache. Storage consistency is enforced by hardware.
- **WIM = 100**
  1. Data may be cached.
  2. Loads for which the target location is in the cache may use that copy of the location.
  3. Stores must be written to main storage. The target location of the store may be cached and must be updated if there.
  4. Exclusive ownership of the block containing the target location is not required for store accesses and consistency operations for the block may be ignored when fetching the block, storing it back, or changing its state from shared to exclusive.
- **WIM = 101**
  1. Data may be cached.
  2. Loads for which the target location is in the cache may use that copy of the location.
  3. Stores must be written to main storage. The target location of the store may be cached and must be updated if there.
  4. Exclusive ownership of the block containing the target location is required before store accesses are allowed. When fetching the block, the processor must indicate that consistency is to be enforced on the bus transaction. If the state of the block is read shared, the processor must gain exclusive use of the block before storing into it.
- **WIM = 110**

  This mode would represent memory that is write through, cache inhibited, and memory coherence not required. This mode is **not supported**.
- **WIM = 111**

  This mode would represent memory that is write through, cache inhibited, and memory coherence required. This mode is **not supported**.

4.8.3 Mismatched WIMG Bits

Accesses to the same storage location using two effective addresses for which the Write Through mode (W bit) differs must meet the memory coherence requirements described in Book II, *PowerPC Virtual Environment Architecture*.

Engineering Note

If an implementation uses a "MESI" coherency protocol, a store addressed to a Write Through page may find the addressed cache block in the cache and modified. If so, the store should update the location in both the cache block and main storage (the normal operation of a store to Write Through storage). It is acceptable for the implementation to write the block back to main storage, in which case it can change the state to "unmodified." It is also acceptable for the implementation to leave the state of the cache block "modified" after updating the location in cache and main storage.

4.9 Reference and Change Recording

If address translation is enabled ($MSR_{IR}=1$ or $MSR_{DR}=1$), reference (R) and change (C) bits are maintained in the Page Table Entry for each real page for accesses due to segment and page table address translation. Reference and change recording is not performed for translations due to BAT or for direct-store (T = 1) segments.

The R and C bits are set automatically by hardware or by software assist in conjunction with normal Page Table processing as follows:

Reference bit

As a result of page table processing for a storage access (load, store, or cache instruction, or instruction fetch), the reference bit may be set to 1 immediately or its setting may be delayed until the storage access is determined to be successful. If the reference bit is not set because the access failed, the implementation must set the reference bit on the next successful access.

The reference bit is only a hint to the operating system about the activity of a page. The reference bit may be set to 1 even though the access was not logically required by the program or was denied by storage protection. Examples of this include:

- Prefetching of instructions that are not subsequently executed.
- Speculative "load" instructions that are subsequently abandoned.
- String operations that specify a length of 0.
- Accesses that cause exceptions and are not completed.

Change Bit

Whenever a data store is executed successfully, as part of the TLB look-up procedure the change bit in the TLB is checked. If it is already set to 1, no further action is taken. If the TLB change bit is 0, it is set to 1 and the corresponding change bit in the Page Table Entry is set to 1.

PowerPC Architecture requires that the Change bit be set to 1 if and only if the store is allowed by storage protection and is logically required by the program.

Execution of either of the *Data Cache Block Touch* instructions (***dcbt***, ***dcbtst***) may result in setting the R bit for a page. Neither instruction may result in setting the C bit for a page.

See section 4.12, "Table Update Synchronization Requirements" on page 53 for the rules software must follow when updating the reference and change bits in the Page Table.

Architecture Note

If the reference and change bits are updated by hardware, this is not necessarily done with atomic read/modify/write operations.

Programming Note

On systems with Translation Lookaside Buffers, the reference and change bits are only set on the basis of TLB activity. When software resets these bits to zero it must synchronize the TLB's actions by invalidating the TLB entries associated with the pages whose reference and change bits were reset.

Engineering Note

Since most TLB reloads do not require setting the reference or change bit, it is suggested that on a TLB miss, the search for the entry be done without fetching the page table entries (PTE's) for exclusive access. This will reduce cache thrashing due to TLB reloads. It is assumed that a nonexclusive request for a PTE will be returned with exclusive access if no other processor has a copy.

4.10 Storage Protection

The storage protection mechanism provides a means for selectively granting read access, granting read/write access, and prohibiting access to areas of storage based on a number of control criteria.

Since the protection mechanism operates as part of the address translation mechanism, storage protection applies to translated accesses only. Instruction storage access protection is active only when $MSR_{IR} = 1$. Data storage access protection is active only when $MSR_{DR} = 1$.

A page (4 KB) crossing is relevant to performance and instruction restart when it corresponds to a protection boundary. Crossing a 4 KB boundary in an area mapped by Block Address Translation or in a direct-store segment should have no effect on performance and should not cause an instruction restart.

For ordinary translated accesses to memory via the Page Table, the Page Protection mechanism described in the next section is active. Different mechanisms are used for Block Address Translation (BAT) accesses (see section 4.10.2, "BAT Protection") and for Direct-store segments (see section 4.6.2, "Direct-store segment protection" on page 38).

4.10.1 Page Protection

The page protection mechanism provides protection at the granularity of a page (4 KB). It is controlled by the following inputs:

- MSR_{PR}, which distinguishes between supervisor state and problem state.
- K_s and K_p, supervisor and problem key bits in the Segment Table Entry or Segment Register.
- PP bits in the Page Table Entry.

A reference made via the segmented address translation mechanism is associated with a Segment Table Entry (STE) and a Page Table Entry (PTE) by the address translation mechanism. The K bits, the PP bits, and the MSR_{PR} bit are used as follows:

A Key value is developed according to the following formula:

$$Key \leftarrow (K_p \ \& \ MSR_{PR}) \ | \ (K_s \ \& \ \neg MSR_{PR})$$

Using the generated Key, the following table is applied:

Key	pp	Page Type	Load Access Permitted	Store Access Permitted
0	00	read/write	yes	yes
0	01	read/write	yes	yes
0	10	read/write	yes	yes
0	11	read only	yes	no
1	00	no access	no	no
1	01	read only	yes	no
1	10	read/write	yes	yes
1	11	read only	yes	no

Key Key selected by state of MSR_{PR} bit
pp PTE page protect bits

Figure 29. Protection Key Processing

When a reference is not permitted because of the protection mechanism one of the following occurs.

- Data Storage interrupt is generated and bit 4 of the DSISR is set to 1.
- Instruction Storage interrupt is generated and bit 4 of the SRR1 is set to 1.

Programming Note

A store that is not permitted because of the storage protection mechanism will not cause a change bit to be set in a PTE; such an access *may* cause a reference bit to be set in a PTE.

4.10.2 BAT Protection

The BAT protection mechanism operates on an entire BAT area, not on individual pages. If an Effective Address is determined to be within a BAT area, the operations described above in section 4.10.1, "Page Protection" are performed, with these exceptions:

- The K_s and K_p bits from the upper BAT register are used, not bits from a Segment Table Entry or Segment Register.
- The PP bits from the upper BAT register are used, not bits from a Page Table Entry.

4.11 Storage Control Instructions

4.11.1 Cache Management Instructions

This section contains the only privileged cache management instruction and additional specifications for the other cache management instructions described in Book II, *PowerPC Virtual Environment Architecture*. See that document for further details.

When data relocate is off, MSR $_{DR}$ = 0, the *Data Cache Block set to Zero* instruction establishes a block in the cache and may not verify that the real address is valid. If a block is created for an invalid real address, a Machine Check may result when an attempt is made to write that block back to storage. The block could be written back as the result of the execution of an instruction that causes a cache miss and the invalid address block is the target for replacement or as the result of a *Data Cache Block Store* instruction.

Data Cache Block Invalidate X-form

dcbi RA,RB

31	///	RA	RB	470	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The action taken is dependent on the storage mode associated with the target, and the state of the block. The list below describes the action to take if the block containing the byte addressed by EA is or is not in the cache.

1. **Coherence Not Required**
   Unmodified Block
   Invalidate the block in the local cache.
   Modified Block
   Invalidate the block in the local cache. (Discard the modified contents.)
   Absent Block
   No action is taken.
2. **Coherence Required**
   Unmodified Block
   Invalidate copies of the block in the caches of all processors.
   Modified Block
   Invalidate copies of the block in the caches of all processors. (Discard the modified contents.)
   Absent Block
   If copies are in the caches of any other processor, cause the copies to be invalidated. (Discard any modified contents.)

When data address translation is enabled, MSR$_{DR}$ = 1, and the virtual address has no translation a Data Storage Interrupt occurs. See 5.5.3, "Data Storage Interrupt" on page 61.

The function of this instruction is independent of the Write Through and Caching Inhibited/Allowed modes of the block containing the byte addressed by EA.

This instruction is treated as a store to the addressed byte with respect to address translation and protection. The reference bit for EA may be set, the reference and change bits may be set, or neither may be set.

If EA specifies a storage address for which T = 1, the instruction is treated as a no-op.

This instruction is privileged.

Special Registers Altered:
None

4.11.2 Segment Register Manipulation Instructions

Move To Segment Register X-form

mtsr SR,RS

31	RS	/	SR	///	210	/
0	6	11	12	16	21	31

SEGREG(SR) ← (RS)

The contents of register RS is placed into Segment Register SR.

This instruction is privileged.

This instruction is defined only for 32-bit implementations. Using it on a 64-bit implementation will cause an Illegal Instruction type Program interrupt.

Special Registers Altered:
None

Move From Segment Register X-form

mfsr RT,SR

31	RT	/	SR	///	595	/
0	6	11	12	16	21	31

RT ← SEGREG(SR)

The contents of Segment Register SR is placed into register RT.

This instruction is privileged.

This instruction is defined only for 32-bit implementations. Using it on a 64-bit implementation will cause an Illegal Instruction type Program interrupt.

Special Registers Altered:
None

Programming Note

For a discussion of software synchronization requirements when altering Segment Registers, please refer to Appendix F, "Synchronization Requirements for Special Registers" on page 83.

Move To Segment Register Indirect X-form

mtsrin RS,RB

[Power mnemonic: mtsri]

31	RS	///	RB	242	/
0	6	11	16	21	31

SEGREG($(RB)_{0:3}$) ← (RS)

The contents of register RS are copied to the Segment Register selected by bits 0:3 of register RB.

This instruction is privileged.

This instruction is defined only for 32-bit implementations. Using it on a 64-bit implementation will cause an Illegal Instruction type Program interrupt.

Special Registers Altered:
None

Move From Segment Register Indirect X-form

mfsrin RT,RB

31	RT	///	RB	659	/
0	6	11	16	21	31

RT ← SEGREG($(RB)_{0:3}$)

The contents of the Segment Register selected by bits 0:3 of register RB are copied into register RT.

This instruction is privileged.

This instruction is defined only for 32-bit implementations. Using it on a 64-bit implementation will cause an Illegal Instruction type Program interrupt.

Special Registers Altered:
None

Programming Note

The RA field is not defined for the ***mtsrin*** and ***mfsrin*** instructions in this architecture. However, ***mtsrin*** and ***mfsrin*** will perform the same function in PowerPC as do ***mtsri*** and ***mfsri*** in Power if RA is 0 in the Power instructions.

4.11.3 Lookaside Buffer Management Instructions (Optional)

While the PowerPC Architecture describes logically separate instruction fetch and fixed-point (including effective address computation) execution units, the programming model is that there is one translation mechanism and, for 32-bit implementations, one set of segment registers.

For performance reasons, most implementations will implement a Segment Lookaside Buffer (64-bit implementations) and a Translation Lookaside Buffer. These are caches of portions of the Segment Table and Page Table respectively. As changes are made to the address translation tables, it is necessary to force the SLB and TLB into line with the updated tables. This is done by invalidating SLB and TLB entries, or occasionally by invalidating the entire SLB or TLB, and allowing the translation caching mechanism to re-fetch from the tables.

Each PowerPC implementation which has an SLB **must** provide means for doing the following:

- Invalidating an individual SLB entry
- Invalidating the entire SLB

Each PowerPC implementation which has a TLB **must** provide means for doing the following:

- Invalidating an individual TLB entry
- Invalidating the entire TLB

An implementation may choose to provide one or more of the instructions listed in this section in order to satisfy requirements in the preceding list. If an instruction is implemented that matches the semantics of an instruction described here, the implementation should be as specified here. Alternatively, an algorithm may be given that performs one of the functions listed above (a loop invalidating individual SLB entries may be used to invalidate the entire SLB, for example), or instructions with different semantics may be implemented. Such algorithms or instructions must be described in Book IV, *PowerPC Implementation Features*.

It is permissible for an instruction described here to be implemented so that more is done than absolutely required. For example, an instruction whose semantics are to purge an SLB entry may be implemented so as to purge an entire congruence class or perhaps even the entire SLB. Such additional actions should be described in Book IV.

If the implementation does not implement an SLB, it does not provide the optional instructions that affect the SLB (***slbie***, ***slbiex***, and ***slbia***). In such an implementation, it is permissible to treat these SLB instructions as no-ops. Similarly, if the implementation does not implement a TLB, it does not provide the optional instructions that affect the TLB (***tlbie***, ***tlbiex***, ***tlbia***, and ***tlbsync***). In such an implementation, it is permissible to treat these TLB instructions as no-ops.

Engineering Notes

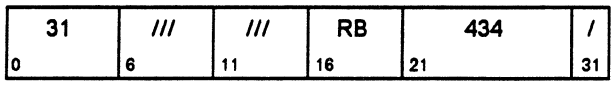
1. It is possible for the hardware to implement more than one set of Segment Registers, such as one for data and one for instructions. If this approach is taken, it is the responsibility of the hardware to keep all sets of registers consistent.
2. It is possible that the hardware implements separate TLB arrays. In this case the size, shape and values contained may be different.
3. If separate TLB arrays are implemented for data and instructions, the requirement for an instruction that purges a TLB entry may be met with a single instruction for both arrays or separate instructions for each array.

Programming Note

Because the presence, absence, and exact semantics of the various Lookaside Buffer management instructions are model dependent, it is recommended that system software "encapsulate" uses of such instructions into subroutines to minimize the impact of moving from one implementation to another.

SLB Invalidate Entry X-form

slbie RB




```
EA ← (RB)
if SLB entry exists for EA, then
  SLB entry ← invalid
```

Let the effective address (EA) be the contents of register RB. If the Segment Lookaside Buffer (SLB) contains an entry corresponding to EA, that entry is made invalid (i.e., removed from the SLB).

The SLB search is done regardless of the settings of MSR_{IR} and MSR_{DR}.

Block Address Translation for EA, if any, is ignored.

This instruction is privileged.

This instruction is optional in PowerPC Architecture.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause an Illegal Instruction type Program interrupt.

Special Registers Altered:

None

Architecture Note

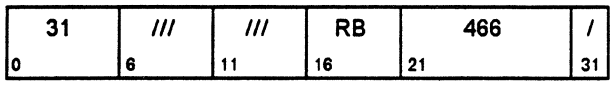
Bits 11:15 of this instruction (ordinarily the position of an RA field) must be zero. This provides implementations the option of using (RA|0) + (RB) address arithmetic for this instruction.

Programming Note

It is not necessary that the ASR point to a valid Segment Table when issuing ***slbie***.

SLB Invalidate Entry by Index X-form

slbiex RB




```
n ← (RB)
SLB entry n ← invalid
```

Let *n* be the contents of register RB. The *n*th SLB entry is made invalid (i.e., removed from the SLB).

The SLB entry is invalidated regardless of the settings of MSR_{IR} and MSR_{DR}.

If the *n*th SLB does not exist, the results are implementation-dependent.

This instruction is privileged.

This instruction is optional in PowerPC Architecture.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause an Illegal Instruction type Program interrupt.

Special Registers Altered:

None

Programming Notes

How software "knows" which SLB entry number is associated with which Segment Table entry, or even how many SLB entries there are, is not specified in the architecture. This must be described in Book IV, *PowerPC Implementation Features*.

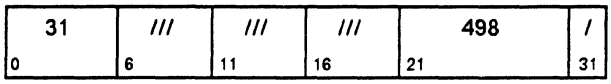
It is not necessary that the ASR point to a valid Segment Table when issuing ***slbiex***.

Architecture Note

Bits 11:15 of this instruction (ordinarily the position of an RA field) must be zero. This provides implementations the option of using (RA|0) + (RB) address arithmetic for this instruction.

SLB Invalidate All X-form

slbia




```
All SLB entries ← invalid
```

The entire SLB is made invalid (i.e., all entries are removed).

The SLB is invalidated regardless of the settings of MSR_{IR} and MSR_{DR}.

This instruction is privileged.

This instruction is optional in PowerPC Architecture.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause an Illegal Instruction type Program interrupt.

Special Registers Altered:
None

Programming Note

It is not necessary that the ASR point to a valid Segment Table when issuing ***slbia***.

TLB Invalidate Entry X-form

tlbie RB

[Power mnemonic: tlbi]

31	///	///	RB	306	/
0	6	11	16	21	31

```
EA ← (RB)
if TLB entry exists for EA, then
  TLB entry ← invalid
```

Let the effective address (EA) be the contents of register RB. If the Translation Lookaside Buffer (TLB) contains an entry corresponding to EA, that entry is made invalid (i.e., removed from the TLB).

The TLB search is done regardless of the settings of MSR_{IR} and MSR_{DR}.

Block Address Translation for EA, if any, is ignored.

If the Segment Register or Segment Table Entry for EA specifies T=1 (a direct-store segment), it is implementation-dependent whether any TLB entries are invalidated and whether the operation is broadcast.

If an implementation supports broadcast of TLB entry invalidates, then:

- The ***tlbie*** instruction(s) must be contained in a critical section, controlled by software locking, so that ***tlbie*** is issued on only one processor at a time.
- A ***sync*** instruction must be issued at the end of the critical section. This will cause the hardware to wait for the effects of the preceding ***tlbie*** instructions(s) to propagate to all processors.
- A processor receiving a ***tlbie*** broadcast will
  1. Prevent execution of any new storage instructions (loads, stores, cache control, reference and change recording, ***tlbie***, ***tlbiex***).
  2. Wait for completion of any outstanding storage instructions, including updates to the reference and change bits associated with the invalidated entry.
  3. Perform the requested TLB invalidation.
  4. Resume normal execution.

This instruction is privileged.

This instruction is optional in PowerPC Architecture.

Special Registers Altered:

None

Architecture Note

Bits 11:15 of this instruction (ordinarily the position of an RA field) must be zero. This provides implementations the option of using (RA|0) + (RB) address arithmetic for this instruction.

Programming Notes

Nothing is guaranteed about instruction fetching in other processors if ***tlbie*** deletes the TLB entry for the page in which some other processor is currently executing.

TLB Invalidate Entry by Index X-form

tlbiex RB

31	///	///	RB	338	/
0	6	11	16	21	31

```
n ← (RB)
TLB entry n ← invalid
```

Let *n* be the contents of register RB. The *n*th TLB entry is made invalid (i.e., removed from the TLB).

The TLB entry is invalidated regardless of the settings of MSR_{IR} and MSR_{DR}.

If the *n*th SLB does not exist, the results are implementation-dependent.

If an implementation supports broadcast of TLB entry invalidates, then:

- The ***tlbiex*** instruction(s) must be contained in a critical section, controlled by software locking, so that ***tlbiex*** is issued on only one processor at a time.
- A ***sync*** instruction must be issued at the end of the critical section. This will cause the hardware to wait for the effects of the preceding ***tlbiex*** instructions(s) to propagate to all processors.
- A processor receiving a ***tlbiex*** broadcast will
  1. Prevent execution of any new storage instructions (loads, stores, cache control, reference and change recording, ***tible***, ***tlbiex***).
  2. Wait for completion of any outstanding storage instructions, including updates to the reference and change bits associated with the invalidated entry.
  3. Perform the requested TLB invalidation.
  4. Resume normal execution.

This instruction is privileged.

This instruction is optional in PowerPC Architecture.

Special Registers Altered:

None

Architecture Note

Bits 11:15 of this instruction (ordinarily the position of an RA field) must be zero. This provides implementations the option of using (RA|0) + (RB) address arithmetic for this instruction.

Programming Notes

How software "knows" which TLB entry number is associated with which Page Table entry, or even how many TLB entries there are, is not specified in the architecture. This must be described in Book IV, *PowerPC Implementation Features.*

It is not necessary that the ASR point to a valid Segment Table or that SDR 1 point to a valid Page Table when issuing ***tlbiex***.

Nothing is guaranteed about instruction fetching in other processors if ***tlbie*** deletes the TLB entry for the page in which some other processor is currently executing.

TLB Invalidate All X-form

tlbia

31	///	///	///	370	/
0	6	11	16	21	31

```
All TLB entries ← invalid
```

The entire TLB is invalidated (i.e., all entries are removed).

The TLB is invalidated regardless of the settings of MSR_{IR} and MSR_{DR}.

This instruction is privileged.

This instruction is optional in PowerPC Architecture.

Special Registers Altered:
None

Programming Notes

It is not necessary that the ASR point to a valid Segment Table or that SDR 1 point to a valid page table when issuing ***tlbia***.

Nothing is guaranteed about instruction fetching in other processors if ***tlbie*** deletes the TLB entry for the page in which some other processor is currently executing.

TLB Synchronize X-form

tlbsync

31	///	///	///	566	/
0	6	11	16	21	31

The ***tlbsync*** instruction waits until all previous ***tlbie***, ***tlbiex***, and ***tlbia*** instructions executed by the processor executing this instruction have been received and completed by all other processors.

This instruction is privileged.

This instruction is optional in PowerPC Architecture, but it must be implemented if any of the following are true:

- A TLB invalidation instruction that broadcasts is implemented.
- The ***eciwx*** or ***ecowx*** instructions are implemented.

Special Registers Altered:
None

4.12 Table Update Synchronization Requirements

This section describes the steps that software must take when updating the tables involved in address translation. Updates to these tables include:

- Adding a new Page Table Entry (PTE).
- Modifying an existing PTE, including the special case of modifying the PTE's Reference bit.
- Deleting a PTE.
- Adding a new Segment Table Entry (STE).
- Modifying an existing STE.
- Deleting a STE.

In a multiprocessor system it is critical that these rules be followed to ensure that all processors see a consistent set of tables. Even in a uniprocessor system certain rules must be followed, notably those regarding Reference and Change bit updates, because software changes must be synchronized with automatic updates by the hardware.

The ***sync*** instruction ensures that all previous TLB invalidate instructions executed by the processor executing the ***sync*** instruction have completed on that processor. However, ***sync*** does not ensure that those invalidate instructions have completed on other processors. A ***tlbsync*** followed by a ***sync*** must be executed to ensure that all previous TLB invalidates executed by the processor executing the synchronizing instructions have been completed on *all* processors.

4.12.1 Page Table Updates

HTAB entries must be locked on multiprocessors. Access to HTAB entries must be appropriately synchronized by software locking of (i.e., guaranteeing exclusive access to) entries or groups of entries if more than one processor can modify the table at once.

On uniprocessors, HTAB entries need not be locked. To adapt the examples given below for the uniprocessor case, simply delete the "lock()" and "unlock()" lines. The ***sync*** instructions shown are still required even on uniprocessors.

TLBs are non-coherent caches of the HTAB. TLB entries must be flushed explicitly with one of the TLB invalidate instructions. The ***sync*** instruction waits until all prior TLB invalidates by this processor are complete. This may cost a ***sync*** per HTAB entry update.

Unsynchronized lookups in the HTAB continue even while it is being modified. Any processor, even including the processor modifying the HTAB, may look in the HTAB at any time in an attempt to reload a TLB entry. An inconsistent HTAB entry must never accidentally become visible, thus there must be synchronization between modifications to the **valid** bit and any other modifications. This costs as many as two ***sync***s per HTAB entry update.

Processors write Reference and Change bits with unsynchronized atomic byte stores. This requires that the V, R, and C bits be in distinct bytes. It also requires extreme care to ensure that no store overwrites one of these bytes accidentally.

In the examples below,

- "lock()" and "unlock()" refer to software locks for exclusive access to the table entry in question,
- **sync** refers to the ***sync*** instruction, and
- **tlbie** refers to the ***tlbie*** instruction.

4.12.1.1 Adding a Page Table Entry

This is the simplest Page Table case. It requires no synchronization with the hardware, just a lock on the PTE in a multiprocessor system. We fill in the entries in the PTE except for the Valid bit, issue a ***sync*** to ensure that the updates have all made it to storage, and turn on the Valid bit.

```
lock(PTE)
PTE_VSID,H,API ← new values
PTE_RPN,R,C,WIM,PP ← new values
sync
PTE_V ← 1
unlock(PTE)
```

4.12.1.2 Modifying a Page Table Entry

General case

In this case a currently-valid PTE must be changed. To do this we must lock the PTE, mark it invalid, flush it from the TLB, update the information in the PTE, mark it valid again, and unlock, using ***sync*** at appropriate times to wait for modifications to complete.

```
lock(PTE)
PTE_V ← 0
sync
tlbie(PTE)
sync
tlbsync
sync
PTE_VSID,H,API ← new values
PTE_RPN,R,C,WIM,PP ← new values
sync
PTE_V ← 1
unlock(PTE)
```

Resetting the Reference bit

In the case where the PTE is modified only to set the Reference bit to 0, a much simpler algorithm suffices because the Reference bit need not be maintained exactly.

```
lock(PTE)
oldR ← PTE_R
if oldR = 1 then
  PTE_R ← 0
  tlbie(PTE)
unlock(PTE)
```

Since only the R and C bits are modified by hardware, and since R and C are in different bytes, the R bit can be set to 0 by reading the current contents of the byte in the PTE containing R (bits 48:55 of the second doubleword on 64-bit implementations, bits 16:23 of the second word on 32-bit implementations), ANDing the value with 0xFE, and storing the byte back into the PTE.

Modifying the virtual address

If the virtual address is being changed to a different address within the same TLB hash class and cache hash class, it suffices to:

```
lock(PTE)
val ← PTE_VSID,API,H,V
insert new VSID into val
PTE_VSID,API,H,V ← val
sync
tlbie(PTE)
sync
tlbsync
sync
unlock(PTE)
```

Here we take advantage of the fact that the store into the first doubleword of the PTE (word, on 32-bit systems) is performed atomically.

Note that if the new address is not a cache synonym of the old, it will be necessary to flush or invalidate the page in the cache(s) as well. This may involve assigning a temporary virtual address that is such a synonym, and using that address to do the cache operations.

4.12.1.3 Deleting a Page Table Entry

Here we just lock the entry, mark it invalid, wait for the change to complete, and unlock.

```
lock(PTE)
PTE_V ← 0
sync
tlbie(PTE)
sync
tlbsync
sync
unlock(PTE)
```

4.12.2 Segment Table Updates

These updates are similar to Page Table updates, but without the complication of hardware updates to Reference and Change bits.

STAB entries must be locked on multiprocessors. Access to STAB entries must be appropriately synchronized by software locking of (i.e., guaranteeing exclusive access to) entries or groups of entries if more than one processor can modify the table at once.

On uniprocessors, STAB entries need not be locked. To adapt the examples given below for the uniprocessor case, simply delete the "lock()" and "unlock()" lines. The ***sync*** instructions shown are still required even on uniprocessors.

SLBs are non-coherent caches of the STAB. SLB entries must be flushed explicitly with one of the SLB invalidate instructions. The ***sync*** instruction waits until all prior SLB invalidates by this processor are complete. This may cost a ***sync*** per STAB entry update.

Unsynchronized lookups in the STAB continue even while it is being modified. Any processor, even including the processor modifying the STAB, may look in the STAB at any time in an attempt to reload a SLB entry. An inconsistent STAB entry must never accidentally become visible, thus there must be synchronization between modifications to the **valid** bit and any other modifications. This costs as many as two ***sync***s per STAB entry update.

In the examples below,

- "lock()" and "unlock()" refer to software locks for exclusive access to the table entry in question,
- **sync** refers to the ***sync*** instruction, and
- **slbie** refers to the ***slbie*** instruction.

4.12.2.1 Adding a Segment Table Entry

We fill in the entries in the STE except for the Valid bit, issue a ***sync*** to ensure that the updates have all made it to storage, and turn on the Valid bit.

```
lock(STE)
STE_ESID,T,Ks,Kp ← new values
if T = 0
  then STE_VSID ← new value
  else STE_IO ← new value
sync
STE_V ← 1
unlock(STE)
```

4.12.2.2 Modifying a Segment Table Entry

In this case a currently-valid STE must be changed. To do this we must lock the STE, mark it invalid, flush it from the SLB, update the information in the STE, mark it valid again, and unlock, using ***sync*** at appropriate times to wait for modifications to complete.

```
lock(STE)
STE_V ← 0
sync
slbie(STE)
sync
STE_ESID,T,Ks,Kp ← new values
if T = 0
  then STE_VSID ← new value
  else STE_IO ← new value
sync
STE_V ← 1
unlock(STE)
```

4.12.2.3 Deleting a Segment Table Entry

Here we just lock the entry, mark it invalid, wait for the change to complete, and unlock.

```
lock(STE)
STE_V ← 0
sync
slbie(STE)
sync
unlock(STE)
```

4.12.3 Segment Register Updates

On an implementation that provides Segment Registers rather than a Segment Table, there is no table to be locked but there are certain synchronization requirements that must be satisfied when using the *Move to Segment Register* instructions. See Appendix F, "Synchronization Requirements for Special Registers" on page 83.

Chapter 5. Interrupts

5.1 Overview

The PowerPC architecture provides an interrupt mechanism to allow the processor to change state as a result of external signals, errors, or unusual conditions arising in the execution of instructions.

System Reset and Machine Check interrupts are not ordered. All other interrupts are ordered such that only one interrupt is reported, and when it is processed (taken), no program state is lost. Since save/restore registers SRR 0 and SRR 1 are serially reusable resources used by most interrupts, program state will be lost when an unordered interrupt is taken.

5.2 Interrupt Synchronization

When an interrupt occurs, SRR 0 is set to point to an instruction such that all preceding instructions have completed execution, no subsequent instruction has begun execution, and the instruction addressed by SRR 0 may or may not have completed execution, depending on the interrupt type.

All interrupts are context synchronizing, as defined in Section 1.7.1, "Context Synchronization" on page 3, except that System Reset and Machine Check interrupts need not be context synchronizing if they are not recoverable (i.e., if bit 30 of SRR 1 is set to 0 by the interrupt).

5.3 Interrupt Classes

Interrupts are classified by whether they are directly caused by the execution of an instruction or are caused by some other system exception. Those that are "system-caused" are:

- System Reset
- Machine Check
- External
- Decrementer

External and Decrementer are maskable interrupts. While $MSR_{EE}=0$, the interrupt mechanism ignores the exceptions that generate these interrupts. Therefore, software may delay the generation of these interrupts by setting $MSR_{EE}=0$ or by failing to set $MSR_{EE}=1$ after processing an interrupt. When any interrupt is taken, MSR_{EE} is set to 0 by the interrupt mechanism, delaying the recognition of any further exceptions causing these interrupts.

System Reset and Machine Check exceptions are not maskable. These exceptions will be recognized regardless of the setting of the MSR.

"Instruction-caused" interrupts are further divided into two classes, *precise* and *imprecise*.

5.3.1 Precise Interrupt

Except for the Imprecise Mode Floating-Point Enabled Exception interrupt, all instruction-caused interrupts are precise. When the execution of an instruction causes a precise interrupt, the following conditions exist at the interrupt point:

1. SRR 0 addresses either the instruction causing the exception or the immediately following instruction. Which instruction is addressed can be determined from the interrupt type and status bits.
2. An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing processor. However, some storage accesses generated by these preceding instructions may not have been performed with respect to all other processors and mechanisms.
3. The instruction causing the exception may not have begun execution, may have partially completed, or may have completed, depending on the interrupt type.
4. Architecturally, no subsequent instruction has begun execution.

5.3.2 Imprecise Interrupt

This architecture defines one imprecise interrupt:

- Imprecise Mode Floating-Point Enabled Exception

When the execution of an instruction causes an imprecise interrupt, the following conditions exist at the interrupt point:

1. SRR 0 addresses either the instruction causing the exception or some instruction following the instruction causing the exception that generated the interrupt.
2. An interrupt is generated such that all instructions preceding the instruction addressed by SRR 0 appear to have completed with respect to the executing processor.
3. If the imprecise interrupt is forced, by the context synchronizing mechanism, due to an instruction that causes another interrupt (e.g., Alignment, DSI) then SRR 0 addresses the interrupt-forcing instruction, and the interrupt-forcing instruction may have been partially executed (see section 5.6, "Partially Executed Instructions" on page 66).
4. If the imprecise interrupt is forced, by the context synchronizing mechanism, due to a context synchronizing instruction (e.g., ***isync***), then SRR 0 addresses the interrupt-forcing instruction, and the interrupt-forcing instruction appears not to have begun execution (except for its forcing the imprecise interrupt).
5. If the imprecise interrupt is not forced by the context synchronizing mechanism, then the instruction addressed by SRR 0 appears not to have begun execution, if it is not the excepting instruction.
6. No instruction following the instruction addressed by SRR 0 appears to have begun execution.

All Floating-Point Enabled Exception interrupts are maskable using the MSR bits FE0 and FE1. Although these interrupts are maskable, they differ significantly from the other maskable interrupts in that the masking of these interrupts is usually controlled by the application program whereas the masking of External and Decrementer interrupts is controlled by the operating system.

Architecture Note

An implementation may define one or more additional interrupts to be imprecise. If this is done, then a complete description of how such imprecise interrupts are implemented by the processor and how they are to be handled by the operating system can be found in the Book IV, *PowerPC Implementation Features* document for the implementation. Such an implementation must provide a means of forcing the processor to process interrupts in a precise fashion as described here, perhaps with reduced performance.

The discussion here assumes that only the Imprecise Mode Floating-Point Enabled Exception interrupt is imprecise.

5.4 Interrupt Processing

Interrupt processing consists of saving a small part of the processor's state in certain registers, identifying the cause of the interrupt in another register, and continuing execution from an address corresponding to the type of interruption. When an exception exists that will cause an interrupt to be generated and it has been determined that the interrupt can be taken, the following actions are performed:

1. SRR 0 is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.
2. Bits 0:15 of SRR 1 are loaded with 16 bits of information specific to the interrupt type.

3. Bits 16:31 of SRR 1 are loaded with a copy of bits 16:31 of the MSR, except for the Machine Check interrupt, for which these bits are set to implementation-dependent values.
4. The MSR is set as described in Figure 30 on page 60. The new values take effect beginning with the first instruction following the interrupt. MSR bits of particular interest are:
   - MSR_{IR} and MSR_{DR} are set to 0 for all interrupt types. Thus relocate is turned off for both instruction fetch and data access beginning with the first instruction following the acceptance of the interrupt. See Chapter 4, "Storage Control" on page 17.
   - MSR_{SF} bit is set to 1 in 64-bit implementations and execution after the interrupt begins in 64-bit mode. This bit is reserved (not defined) in 32-bit implementations.
5. Instruction fetch and execution resumes, using the new MSR value, at a location specific to the interrupt type. The location is determined by adding the interrupt's offset (see Figure 31 on page 60) to the base address determined by MSR_{IP} (see Interrupt Prefix on page 6). For a Machine Check that occurs when $MSR_{ME}=0$, the Checkstop state is entered (the machine stops executing instructions). See 5.5.2, "Machine Check Interrupt" on page 60.

Interrupts do not clear reservations obtained with ***lwarx*** or ***ldarx***. The operating system should do so at appropriate points, such as at process switch.

Programming Note

In some implementations, any instruction fetch with $MSR_{IR}=1$, and any load or store with $MSR_{DR}=1$, may have the side effect of modifying SRRs 0 and 1.

Programming Note

In general, at process switch, due to possible process interlocks and possible data availability requirements, the operating system needs to consider executing the following:

- ***stwcx.***, to clear the reservation if one is outstanding, to ensure that a ***lwarx*** or ***ldarx*** in the "old" process is not paired with a ***stwcx.*** or ***stdcx.*** in the "new" process.
- ***sync***, to ensure that all storage operations of an interrupted process are complete with respect to other processors before that process begins executing on another processor.
- ***isync*** or ***rfi***, to ensure that the instructions in the "new" process execute in the "new" context.

Programming Note

The operating system should manage MSR_{RI} as follows:

- In the Machine Check and System Reset interrupt handlers, interpret SRR 1 bit 30 (where MSR_{RI} is placed) as:
  - 0: interrupt is not recoverable
  - 1: interrupt is recoverable with respect to the processor
- In each interrupt handler, when enough state has been saved that a Machine Check or System Reset interrupt can be recovered from, set MSR_{RI} to 1.
- In each interrupt handler, do the following just before returning.
  - Set MSR_{RI} to 0.
  - Set SRR 0 and SRR 1 to the values to be used by ***rfi***. The new value of SRR 1 should have bit 30 set to 1 (which will happen naturally if SRR 1 is restored to the value saved there by the interrupt, because the interrupt handler will not be executing this sequence unless the interrupt is recoverable).
  - Execute ***rfi***.

Engineering Note

Implementations that use emulation assists must report, in SRR 0 and in the DAR if applicable, the effective addresses computed by the instruction being emulated and not those computed by one of the emulation assist instructions.

5.5 Interrupt Definitions

Figure 30 on page 60 below shows all the types of interrupts and the values assigned to the MSR for each. Figure 31 on page 60 shows the offset of the first instruction, for each interrupt type.

Interrupt Type	MSR bit IP	ME	SF{}
System Reset	-	-	1
Machine Check	-	0	1
Data Storage	-	-	1
Instruction Storage	-	-	1
External	-	-	1
Alignment	-	-	1
Program	-	-	1
FP Unavailable	-	-	1
Decrementer	-	-	1
System Call	-	-	1
Trace	-	-	1
Floating-Point Assist	-	-	1

0 bit is set to 0
1 bit is set to 1
- bit is not altered

Defined bits not shown above (BE, DR, EE, FE0, FE1, FP, IR, PR, RI, and SE) are set to 0.

Reserved bits are set as if written as 0.

In 32-bit implementations, the SF bit (bit 31) is reserved.

Figure 30. MSR Setting Due to Interrupt

Offset (hex)	Interrupt Type
00000	Reserved
00100	System Reset
00200	Machine Check
00300	Data Storage
00400	Instruction Storage
00500	External
00600	Alignment
00700	Program
00800	Floating-Point Unavailable
00900	Decrementer
00A00	Reserved
00B00	Reserved
00C00	System Call
00D00	Trace
00E00	Floating-Point Assist
00E10	Reserved
. . .	. . .
00FFF	Reserved
01000	Reserved, implementation-specific
. . .	. . .
02FFF	(end of interrupt vector locations)

Figure 31. Offset of First Instruction by Interrupt Type

Programming Note

Use of any of the locations shown as reserved risks incompatibility with future implementations.

5.5.1 System Reset Interrupt

System Reset begins with a System Reset interrupt.

If the System Reset exception caused the processor state to be corrupted such that the content of SRR 0 or SRR 1 are not valid or other processor resources are corrupt and would preclude a reliable restart, then the processor sets SRR 1 bit 30 (where MSR_{RI} is normally placed) to 0, to indicate to the interrupt handler that the interrupt is not recoverable.

The following registers are set:

SRR 0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR 1
0:15 Set to 0.
16:29 Loaded from bits 16:29 of the MSR.
30 Loaded from bit 30 of the MSR if the processor is in a recoverable state, otherwise set to 0.
31 Loaded from bit 31 of the MSR.

MSR See Figure 30.

Execution resumes at offset 0x00100 from the base real address indicated by MSR_{IP}.

Engineering Note

Every attempt should be made to allow continuing execution.

5.5.2 Machine Check Interrupt

Machine Check interrupts are enabled when $MSR_{ME}=1$. If $MSR_{ME}=0$ and a Machine Check occurs, the processor enters the Checkstop state.

Disabled Machine Check (Checkstop State)

When a processor is in Checkstop state, instruction processing is suspended and generally cannot be restarted without resetting the processor. Some implementations may freeze the content of all latches when entering Checkstop state so that the state of the processor can be analyzed as an aid in problem determination.

Enabled Machine Check

If the Machine Check exception caused the processor state to be corrupted such that the content of SRR 0 or SRR 1 are not valid or other processor resources are corrupt and would preclude a reliable restart, then the processor sets SRR 1 bit 30 (where MSR_{RI} is

normally placed) to 0, to indicate to the interrupt handler that the interrupt is not recoverable.

In some systems, the operating system may attempt to identify and log the cause of the Machine Check. If the exception that caused the Machine Check does not preclude continued execution (i.e., if SRR 1 bit 30 is set to 1 for the interrupt handler), the processor must be able to continue execution at the Machine Check interrupt vector address.

The following registers are set:

SRR 0 Set on a "best effort" basis to the effective address of some instruction that was executing or was about to be executed when the Machine Check exception occurred. For further details see the Book IV, *PowerPC Implementation Features* document for the implementation.

SRR 1 See the Book IV, *PowerPC Implementation Features* document for the implementation.

MSR See Figure 30 on page 60.

Execution resumes at offset 0x00200 from the base real address indicated by MSR_{IP}.

Programming Note

On some implementations a Machine Check interrupt may occur due to referencing an invalid (nonexistent) real address, either directly (with $MSR_{DR}=0$), or through an invalid translation. On such a system, execution of *Data Cache Block set to Zero* can cause a delayed Machine Check interrupt by introducing a block into the data cache that is associated with an invalid real address. A Machine Check interrupt could eventually occur when and if a subsequent attempt is made to store that block to main storage.

Engineering Note

Not all implementations provide the same level of error checking. The cause of Machine Check is implementation-dependent. Every attempt should be made to allow continuing execution.

5.5.3 Data Storage Interrupt

A Data Storage interrupt occurs when no higher priority exception exists and a data storage access cannot be performed for any of the following reasons:

- The instruction results in a Direct-Store Error exception.
- The effective address of a load, store, ***dcbi***, ***dcbst***, ***dcbf***, ***dcbz***, or ***icbi*** instruction cannot be translated.
- The instruction is not supported for the type of storage addressed. (An interrupt may not occur for this condition; see Section 4.6.3, "Instructions not supported for T=1" on page 38).
- The access violates storage protection.
- Execution of a ***eciwx*** or ***ecowx*** instruction is disallowed because $EAR_E=0$.

Such accesses can be generated by load/store type instructions (discussed in Book I, *PowerPC User Instruction Set Architecture*), certain storage control instructions, certain cache control instructions (discussed in Book II, *PowerPC Virtual Environment Architecture*), and the ***eciwx*** and ***ecowx*** instructions (discussed in Book III, *PowerPC Operating Environment Architecture*).

If a ***stwcx.*** or ***stdcx.*** has an effective address for which a normal *store* would cause a Data Storage interrupt, but the processor does not have the reservation from ***lwarx*** or ***ldarx***, then it is implementation-dependent whether or not a Data Storage interrupt occurs.

If a *Move Assist* instruction has a length of zero (in the XER), a Data Storage interrupt does not occur, regardless of the effective address.

The interrupt cause is defined in the Data Storage Interrupt Status Register. These interrupts also use the Data Address Register.

The following registers are set:

SRR 0 Set to the effective address of the instruction that caused the interrupt.

SRR 1

0:15 Set to 0.

16:31 Loaded from bits 16:31 of the MSR.

MSR See Figure 30 on page 60.

DSISR

0 Set to 1 if a load or store instruction results in a Direct-Store Error exception, otherwise 0.

1 Set to 1 if the translation of an attempted access is not found in the hashed primary HTEG, or in the re-hashed secondary HTEG, or in the range of a DBAT register; otherwise 0.

2:3 Set to 0.

4 Set to 1 if a storage access is not permitted by the page or DBAT protection mechanism described on page 44, otherwise 0.

5 Set to 1 if the access was due to an ***eciwx***, ***ecowx***, ***lwarx***, ***ldarx***, ***stwcx.***, or ***stdcx.*** that addresses a direct-store segment (T=1 in Segment register or Segment Table Entry), or if the access was due to a ***lwarx***, ***ldarx***,

stwcx., or ***stdcx.*** that addresses Write Through storage; set to 0 otherwise.

6 Set to 1 for a store operation and to 0 for a load operation.

7:8 Set to 0.

9 Reserved for DABR (see the Book IV, *PowerPC Implementation Features* document for the implementation).

10 Set to 1 if the Segment Table Search fails to find a translation for the effective address, otherwise set to 0.

11 Set to 1 if execution of a ***eciwx*** or ***ecowx*** instruction was attempted with $EAR_E=0$, otherwise set to 0.

12:31 Set to 0.

DAR Set to the effective address of a storage element as described in the following list.

- A byte in the first word accessed in the page that caused the Data Storage interrupt, for a byte, halfword, or word access to a non-direct-store segment.
- A byte in the first doubleword accessed in the page that caused the Data Storage interrupt, for a doubleword access to a non-direct-store segment.
- A byte in the first word accessed in the BAT area that caused the Data Storage interrupt, for a byte, halfword, or word access to a BAT area.
- A byte in the first doubleword accessed in the BAT area that caused the Data Storage interrupt, for a doubleword access to a BAT area.
- Any effective address in the range of storage being addressed, for a Direct-Store Error exception.

Execution resumes at offset 0x00300 from the base real address indicated by MSR_{IP}.

5.5.4 Instruction Storage Interrupt

An Instruction Storage interrupt occurs when no higher priority exception exists and an attempt to fetch the next instruction to be executed cannot be performed for any of the following reasons:

- The effective address cannot be translated.
- The fetch access is to a direct-store segment.
- The fetch access violates storage protection.

Such accesses can only be generated by instruction fetches. The following registers are set:

SRR 0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, SRR 0 is set to the branch target address).

SRR 1

0 Set to 0.

1 Set to 1 if the translation of an attempted access is not found in the hashed primary HTEG, or in the re-hashed secondary HTEG, or in the range of an IBAT register; otherwise 0.

2 Set to 0.

3 Set to 1 if the fetch access was to a direct-store segment (T=1 in Segment Register or Segment Table Entry); set to 0 otherwise.

4 Set to 1 if a storage access is not permitted by the page or IBAT protection mechanism described on page 44, otherwise 0.

5:9 Set to 0.

10 Set to 1 if the Segment Table Search fails to find a translation for the effective address, otherwise set to 0.

11:15 Set to 0.

16:31 Loaded from bits 16:31 of the MSR.

MSR See Figure 30 on page 60.

Execution resumes at offset 0x00400 from the base real address indicated by MSR_{IP}.

5.5.5 External Interrupt

An External interrupt occurs when no higher priority exception exists, an External interrupt exception is presented to the interrupt mechanism, and $MSR_{EE}=1$. The occurrence of the interrupt does *not* cancel the request.

The following registers are set:

SRR 0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR 1

0:15 Set to 0.

16:31 Loaded from bits 16:31 of the MSR.

MSR See Figure 30 on page 60.

Execution resumes at offset 0x00500 from the base real address indicated by MSR_{IP}.

5.5.6 Alignment Interrupt

An Alignment interrupt occurs when no higher priority exception exists and the implementation cannot perform a storage access for one of the reasons listed below. The term "protection boundary," used below, refers to the boundary between protection domains. A protection domain is a direct-store segment, a block of storage defined by a BAT entry, or a 4K block of storage defined by a Page Table entry. Protection domains are defined only when DR = 1.

- The operand of a floating-point load or store is not word-aligned, for any storage class.
- The operand of a fixed-point doubleword load or store is not word-aligned, for any storage class.
- The operand of ***lmw, stmw, lwarx***, or ***stwcx.*** is not word-aligned, or the operand of ***ldarx*** or ***stdcx.*** is not doubleword-aligned, for any storage class.
- The operand of a floating-point load or store is in a direct-store segment (T = 1).
- The operand of an elementary or string load or store crosses a protection boundary.
- The operand of ***lmw*** or ***stmw*** crosses a segment or BAT boundary.
- The operand of *Data Cache Block set to Zero* (***dcbz***) is in a page that is write through or cache inhibited, for a virtual mode access.

In all cases above, an implementation may correctly do the operation and not cause an Alignment interrupt. Details can be found in the Book IV, *PowerPC Implementation Features* document for the implementation.

Engineering Note

If attempt is made to execute an ***lmw*** or ***stmw*** instruction having an incorrectly aligned effective address, early implementations *must* either correctly transfer the addressed bytes or cause an Alignment interrupt, for reasons of compatibility with the Power Architecture.

The following registers are set:

SRR 0 Set to the effective address of the instruction that caused the interrupt.

SRR 1

0:15 Set to 0.

16:31 Loaded from bits 16:31 of the MSR.

MSR See Figure 30 on page 60.

DSISR

0:11 Set to 0.

12:13 Set to bits 30:31 of the instruction if DS-form.
Set to 0b00 if D- or X-form. (Set to 0b00 on 32-bit implementations.)

14 Set to 0.

15:16 Set to bits 29:30 of the instruction if X-form.
Set to 0b00 if D- or DS-form.

17 Set to bit 25 of the instruction if X-form.
Set to bit 5 of the instruction if D- or DS-form.

18:21 Set to bits 21:24 of the instruction if X-form.
Set to bits 1:4 of the instruction if D- or DS-form.

22:26 Set to bits 6:10 of the instruction (RT/RS/FRT/FRS), except undefined for ***dcbz***.

27:31 Set to bits 11:15 of the instruction (RA) for update form instructions; set to either bits 11:15 of the instruction or to any register number not in the range of registers loaded by a valid form instruction, for ***lmw, lswi***, and ***lswx***; undefined for other instructions.

Engineering Note

The requirement for ***lmw, lswi***, and ***lswx*** assures compatibility with the program that emulates these instructions on the Power architecture. It can be met by storing zeros for ***lmw***, and by storing the RT field with one subtracted from it for ***lmw, lswi*** and ***lswx*** (the *load string* instructions wrap from GPR 31 to 0, so simply storing zeros is not adequate).

DAR Set to the effective address of the data access as computed by the instruction causing the alignment exception.

For an X-form Load or Store, it is acceptable to set the DSISR to the same value that would have resulted if the corresponding D- or DS-form instruction had caused the interrupt. Similarly, for a D- or DS-form Load or Store, it is acceptable to set the DSISR to the value that would have resulted for the corresponding X-form instruction. For example, an unaligned ***lwax*** (that crosses a protection boundary) would normally, following the description above, cause the DSISR to be set to binary:

```
000000000000 00 0 01 0 0101 ttttt ?????
```

where "ttttt" denotes the RT field, and "?????" denotes undefined bits. However, it is acceptable if it causes the DSISR to be set as for ***lwa***, which is

```
000000000000 10 0 00 0 1101 ttttt ?????
```

If there is no corresponding alternate form instruction (e.g., for ***lwaux***), the value described above must be set in the DSISR.

The instruction pairs that may use the same DSISR value are:

lbz/lbzx	lbzu/lbzux	lhz/lhzx	lhzu/lhzux
lha/lhax	lhau/lhaux	lwz/lwzx	lwzu/lwzux
lwa/lwax	ld/ldx	ldu/ldux	
stb/stbx	stbu/stbux	sth/sthx	sthu/sthux
stw/stwx	stwu/stwux	std/stdx	stdu/stdux
lfs/lfsx	lfsu/lfsux	lfd/lfdx	lfdu/lfdux
stfs/stfsx	stfsu/stfsux	stfd/stfdx	stfdu/stfdux

Execution resumes at offset 0x00600 from the base real address indicated by MSR_{IP}.

Programming Note

Software should *not* attempt to obtain a reservation for an unaligned ***lwarx*** or ***ldarx***, nor to simulate an unaligned ***stwcx.*** or ***stdcx.***.

5.5.7 Program Interrupt

An Program interrupt occurs when no higher priority exception exists and one or more of the following exceptions arises during execution of an instruction:

Floating-Point Enabled Exception
A Floating-Point Enabled Exception type Program interrupt is generated when the expression

$$(MSR_{FE0} \mid MSR_{FE1}) \ \& \ FPSCR_{FEX}$$

is 1. $FPSCR_{FEX}$ is turned on by the execution of a floating-point instruction that causes an enabled exception or by the execution of a "Move to FPSCR" type instruction that results in both an exception bit and its corresponding enable bit being 1.

Illegal Instruction
An Illegal Instruction type Program interrupt is generated when execution is attempted of an instruction with an illegal opcode or an illegal combination of opcode and extended opcode fields, or when execution is attempted of an optional instruction that is not provided by the implementation (with the exception of optional instructions that are treated as no-ops). Also, implementations are allowed to generate this interrupt for any invalid form instructions.

See the Book I, *PowerPC User Instruction Set Architecture* appendix "Incompatibilities with the Power Architecture" regarding moving to and from the MQ and Decrementer registers.

Privileged Instruction
A Privileged Instruction type Program interrupt is generated when the execution of a privileged instruction is attempted and $MSR_{PR} = 1$. Some implementations may generate this interrupt for ***mtspr*** or ***mfspr*** with an invalid SPR field if $spr_0 = 1$ and $MSR_{PR} = 1$.

Trap
A Trap type Program interrupt is generated when any of the conditions specified in a *Trap* instruction is met.

The following registers are set:

SRR 0 For all Program interrupts except a Floating-Point Enabled Exception when in one of the Imprecise modes, set to the effective address of the instruction that caused the Program interrupt.

For an Imprecise Mode Floating-Point Enabled Exception, set to the effective address of the excepting instruction or to the effective address of some subsequent instruction. If it points to a subsequent instruction, that instruction has not been executed. If a subsequent instruction is *Synchronize* (***sync***) or *Instruction Synchronize* (***isync***), SRR 0 will not point more than four bytes beyond the ***sync*** or ***isync*** instruction.

If $FPSCR_{FEX} = 1$ but Floating-Point Enabled Exception interrupt is disabled by having both MSR_{FE0} and $MSR_{FE1} = 0$, a Floating-Point Enabled Exception interrupt will occur prior to or at the next synchronizing event if these MSR bits are altered with any instruction that can set the MSR so that the expression

$$(MSR_{FE0} \mid MSR_{FE1}) \ \& \ FPSCR_{FEX}$$

is 1. When this occurs, SRR 0 is loaded with the address of the instruction that would have executed next, not with the address of the instruction that modified the MSR causing the interrupt.

SRR 1

0:10	Set to 0.
11	Set to 1 for a Floating-Point Enabled Exception type Program interrupt, otherwise 0.
12	Set to 1 for an Illegal Instruction type Program interrupt, otherwise 0.
13	Set to 1 for a Privileged Instruction type Program interrupt, otherwise 0.
14	Set to 1 for a Trap type Program interrupt, otherwise 0.
15	Set to 0 if SRR 0 contains the address of the instruction causing the exception, and to 1 if SRR 0 contains the address of a subsequent instruction.
16:31	Loaded from bits 16:31 of the MSR.

Only one of bits 11:14 can be set to 1.

MSR See Figure 30 on page 60.

Execution resumes at offset 0x00700 from the base real address indicated by MSR_{IP}.

Engineering Note

If the Imprecise Recoverable Mode Floating-Point Enabled Exception interrupt is implemented as imprecise, the hardware must provide, at the minimum, the address at which to resume the interrupted process (this is given in SRR 0), the excepting instruction's opcode, extended opcode, and record bit, the source values or registers, and the target register. This information can be provided directly in registers or by means of a pointer to the excepting instruction. The manner in which it is provided is described in the Book IV, *PowerPC Implementation Features* document for the implementation.

5.5.8 Floating-Point Unavailable Interrupt

A Floating-Point Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point loads, stores, and moves), and $MSR_{FP} = 0$.

The following registers are set:

SRR 0 Set to the effective address of the instruction that caused the interrupt.

SRR 1
- **0:15** Set to 0.
- **16:31** Loaded from bits 16:31 of the MSR.

MSR See Figure 30 on page 60.

Execution resumes at offset 0x00800 from the base real address indicated by MSR_{IP}.

5.5.9 Decrementer Interrupt

A Decrementer interrupt occurs when no higher priority exception exists, the Decrementer exception exists, and $MSR_{EE} = 1$. The occurrence of the interrupt cancels the request.

The following registers are set:

SRR 0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR 1
- **0:15** Set to 0.
- **16:31** Loaded from bits 16:31 of the MSR.

MSR See Figure 30 on page 60.

Execution resumes at offset 0x00900 from the base real address indicated by MSR_{IP}.

5.5.10 System Call Interrupt

A System Call interrupt occurs when a *System Call* instruction is executed.

The following registers are set:

SRR 0 Set to the effective address of the instruction following the *System Call* instruction.

SRR 1
- **0:15** Undefined.
- **16:31** Loaded from bits 16:31 of the MSR.

MSR See Figure 30 on page 60.

Execution resumes at offset 0x00C00 from the base real address indicated by MSR_{IP}.

Architecture Note

Bits 0:15 of SRR 1 are set to an undefined value, rather than to 0, because some early implementations may save bits 16:31 of the instruction there.

5.5.11 Trace Interrupt

The Trace interrupt may optionally be implemented.

If implemented, a Trace interrupt occurs when no higher priority exception exists and either $MSR_{SE} = 1$ and any instruction except ***rfi*** is successfully completed, or $MSR_{BE} = 1$ and a branch instruction is completed.

The following registers are set:

SRR 0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR 1
- **0:15** See the Book IV, *PowerPC Implementation Features* document for the implementation.
- **16:31** Loaded from bits 16:31 of the MSR.

MSR See Figure 30 on page 60.

For further details see the Book IV, *PowerPC Implementation Features* document for the implementation.

Execution resumes at offset 0x00D00 from the base real address indicated by MSR_{IP}.

5.5.12 Floating-Point Assist Interrupt

The Floating-Point Assist interrupt may optionally be implemented. Its purpose is to allow software assistance for relatively infrequent and complex floating-point operations such as computations involving denormalized numbers.

If implemented, the following registers are set:

SRR 0 Set to the effective address of the instruction that caused the Floating-Point Assist interrupt.

SRR 1

0:15 See the Book IV, *PowerPC Implementation Features* document for the implementation.

16:31 Loaded from bits 16:31 of the MSR.

MSR See Figure 30 on page 60.

For further details see the Book IV, *PowerPC Implementation Features* document for the implementation.

Execution resumes at offset 0x00E00 from the base real address indicated by MSR_{IP}.

5.6 Partially Executed Instructions

The architecture permits certain instructions to be partially executed when an Alignment or Data Storage interrupt occurs, or an imprecise interrupt is forced by an instruction that causes an Alignment or Data Storage exception. These are:

1. *Load Multiple* or *Load String* that causes an Alignment or Data Storage interrupt: Some registers in the range of registers to be loaded may have been loaded.
2. *Store Multiple* or *Store String* that causes an Alignment or Data Storage interrupt: Some bytes of storage in the range addressed may have been updated.
3. An elementary (non-multiple and non-string) store that causes an Alignment or Data Storage interrupt: Some bytes just before the boundary may have been updated. If the instruction normally alters CR0 (***stwcx.***, ***stdcx.***), CR0 is set to an undefined value. For update forms, the update register (RA) is not altered.
4. A floating-point load that causes an Alignment or Data Storage interrupt: the target register may be altered. For update forms, the update register (RA) is not altered.

In the cases above, the questions of how many registers and how much storage is altered are implementation-, instruction-, and boundary-dependent. However, storage protection is not violated. Furthermore, if some of the data accessed is in direct-store (T = 1), and the instruction is not supported for direct-store, the locations in direct-store are not accessed.

In the following situation, partial execution is not allowed (this preserves restartability):

An elementary (non-multiple and non-string) fixed-point load that causes an Alignment or Data Storage interrupt: the target register is not altered. For update forms, the update register (RA) is not altered.

5.7 Exception Ordering

Since multiple exceptions can exist at the same time and the architecture does not provide for reporting more than one interrupt at a time, the generation of more than one interrupt is prohibited. Also some exceptions would be lost if they were not recognized and handled when they occur. For example, if an external interrupt was generated when a data storage exception existed, the data storage exception would be lost. If the data storage exception was caused by a *Store Multiple* instruction that spanned a page boundary and the exception was a result of attempting to access the second page, the store could have modified locations in the first page even though it appeared that the *Store Multiple* instruction was never executed.

In addition, the architecture defines imprecise interrupts that must be recoverable, cannot be lost, and can occur at any time with respect to the executing instruction stream. Some of the maskable and nonmaskable exceptions are persistent and can be deferred. The following exceptions persist even though some other interrupt is generated:

- Floating-Point Enabled Exceptions
- External
- Decrementer

For the above reasons, all exceptions are prioritized with respect to other exceptions that may exist at the same instant to prevent the loss of any exception that is not persistent. Some exceptions cannot exist at the same instant as some others.

5.7.1 Unordered Interrupt Conditions

The exceptions listed here are unordered, meaning that they may occur at any time regardless of the state of the interrupt mechanism. These exceptions must be recognized and processed when presented.

1. System Reset
2. Machine Check

All other interrupts are ordered with respect to the interrupt mechanism resources.

5.7.2 Ordered Exceptions

The exceptions described here are ordered, meaning that only one can be reported. However, the single ordered exception that can be reported may exist in concert with unordered exceptions. Ordered exceptions may or may not be instruction-caused. The two lists identify the ordered interrupts by type. The order within the lists does not imply priority but only lists the possible exceptions that may be reported.

System-caused or Imprecise

1. Program
   - Imprecise Mode Floating-Point Enabled Exception
2. External
3. Decrementer

Instruction-caused and Precise

1. Instruction Storage
2. Program
   - Illegal Instruction
   - Privileged Instruction
3. Function Dependent

   3.a Fixed-Point
   - 1a Program - Trap
   - 1b System Call
   - 1c.1 Alignment
   - 1c.2 Data Storage
   - 2 Trace (if implemented)

   3.b Floating-Point
   - 1 FP Unavailable
   - 2a Program
     - Precise Mode Floating-Point Enabled Excep'n
   - 2b Floating-Point Assist (if implemented)
   - 2c.1 Alignment
   - 2c.2 Data Storage
   - 3 Trace (if implemented)

For implementations that execute multiple instructions in parallel using pipeline or super-scalar techniques, or combinations of these, it can be difficult to understand the ordering of exceptions. To understand this ordering it is useful to consider a model in which an instruction is fetched, decoded, and then executed. In this model, the exceptions a single instruction would generate are in the order shown in the list of instruction-caused exceptions. Exceptions with different numbers have different ordering. Exceptions with the same numbering but different lettering are mutually exclusive and cannot be caused by the same instruction.

Even on processors that are capable of executing several instructions simultaneously, or out of order, instruction-caused interrupts (precise and imprecise) occur in program order.

5.8 Interrupt Priorities

This section describes the relationship of nonmaskable, maskable, precise, and imprecise interrupts. In the following descriptions, the interrupt mechanism waiting for all possible exceptions to be reported includes only exceptions caused by previously initiated instructions (e.g. it does not include waiting for the Decrementer to step through zero). The exceptions are listed in order of highest to lowest priority.

1. System Reset
   System Reset exception has the highest priority of all exceptions. If this exception exists, the interrupt mechanism ignores all other exceptions and generates a System Reset interrupt.

   Once the System Reset interrupt is generated, no nonmaskable interrupts are generated due to exceptions caused by instructions issued prior to the generation of this interrupt.

2. Machine Check
   Machine Check exception is the second highest priority exception. If this exception exists and a System Reset exception does not exist, the interrupt mechanism ignores all other exceptions and generates a Machine Check interrupt.

   Once the Machine Check interrupt is generated, no nonmaskable interrupts are generated due to exceptions caused by instructions issued prior to the generation of this interrupt.

3. Instruction Dependent
   This exception is the third highest priority exception. When this exception is created, the interrupt mechanism waits for all possible Imprecise exceptions to be reported. It then generates the appropriate ordered interrupt if no higher priority interrupt exception exists when the interrupt is to be generated. Within this category a particular instruction may present more than a single exception. When this occurs, those exceptions are ordered in priority as indicated in the following lists.

   A. Fixed-Point Loads and Stores
      a. Alignment
      b. Data Storage
      c. Trace (if implemented)

   B. Floating-Point Loads and Stores
      a. Floating-Point Unavailable
      b. Alignment
      c. Data Storage
      d. Trace (if implemented)

C. Other Floating-Point Instructions

a. Floating-Point Unavailable
b. Program - Precise Mode Floating-Point Enabled Exception
c. Floating-Point Assist (if implemented)
d. Trace (if implemented)

Not all floating-point instructions can cause enabled exceptions.

D. ***rfi*** and ***mtmsr***

a. Program - Privileged Instruction
b. Program - Precise Mode Floating-Point Enabled Exception
c. Trace (if implemented)

If the MSR bits FE0 and FE1 are set such that Precise Mode Floating-Point Enabled Exception interrupts are enabled and the FPSCR(FEX) bit is set, a Program interrupt will result prior to or at the next synchronizing event.

The Trace interrupt should not be generated after an ***rfi***.

E. Other exceptions
These exceptions are mutually exclusive and have the same priority:

- Program - Trap
- System Call
- Program - Privileged Instruction
- Program - Illegal Instruction

F. Instruction Storage
This exception has the lowest priority in this category. It is only recognized when all instructions prior to the instruction causing this exception appear to have completed and that instruction is to be executed.

The priority of this interrupt is specified for completeness and to ensure that it is not given more favorable treatment. It is acceptable for an implementation to treat this interrupt as though it had a lower priority.

4. Program - Imprecise Mode Floating-Point Enabled Exception
This exception is the fourth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

5. External
This exception is the fifth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

6. Decrementer
This exception is the lowest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

Chapter 6. Timer Facilities

6.1 Overview

The Time Base and the Decrementer provide timing functions for the system. Specific instructions are provided for reading and writing the Time Base, while the Decrementer is manipulated as an SPR. Both are volatile resources and must be initialized during start up.

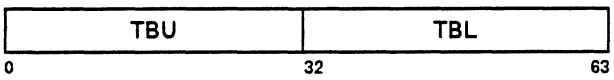
Time Base (TB)
The Time Base provides a long-period counter driven by an implementation-dependent frequency.

Decrementer (DEC)
The Decrementer, a counter that is updated at the same rate as the Time Base, provides a means of signalling an interrupt after a specified amount of time has elapsed unless

- the Decrementer is altered in the interim, **or**
- the Time Base update frequency changes.

6.2 Time Base

The Time Base (TB) is a 64-bit register (see Figure 32) containing a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the low-order bit (bit 63). The frequency at which the counter is updated is implementation dependent and need not be constant over long periods of time.



Field	Description
TBU	Upper 32 bits of Time Base
TBL	Lower 32 bits of Time Base

Figure 32. Time Base

The Time Base runs continuously when powered on. There is no automatic initialization of the Time Base to a known value when the CPU is powered up; system software must perform this initialization if the value of the Time Base at any instant (rather than the difference between two values of the Time Base at different instants) is important.

The Time Base increments until its value becomes 0xFFFF_FFFF_FFFF_FFFF ($2^{64} - 1$). At the next increment, its value becomes 0x0000_0000_0000_0000. There is no interrupt or other indication when this occurs.

The period of the Time Base depends on the driving frequency. As an order of magnitude example, suppose that the CPU clock is 100 MHz and that the Time Base is driven by this frequency divided by 32. Then the period of the Time Base would be

$$T_{TB} = \frac{2^{64} \times 32}{100\ \text{MHz}} = 5.90 \times 10^{12}\ \text{seconds}$$

which is approximately 187,000 years.

The PowerPC Architecture does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock, in a PowerPC system. The Time Base update frequency is not required to be constant.

What *is* required, so that system software can keep time of day and operate interval times, is:

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base changes, plus a means to determine what the current update frequency is, ***or***
- The update frequency of the Time Base is under the control of the system software.

Programming Notes

Assuming that the operating system initializes the Time Base on power-on to some reasonable value and that the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the Time Base will be monotonically increasing. If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

On an implementation that performs speculative execution, the Time Base may be read arbitrarily far "ahead" of the point at which it appears in the instruction stream. If it is important that this not occur, a context synchronizing operation such as the ***isync*** instruction should be placed immediately before the instructions that read the Time Base.

See the description of the Time Base in Book II, *PowerPC Virtual Environment Architecture* for ways to compute time of day in POSIX format from the Time Base.

Architecture Notes

It is intended that the Time Base be useful for timing reasonably short sequences of code (a few hundred instructions) and for low-overhead time stamps for tracing. The Time Base should not "tick" faster than the CPU instruction clock. Driving the Time Base directly from the CPU instruction clock is probably finer granularity than necessary; the instruction clock divided by 8, 16, or 32 would be more appropriate.

The Time Base driving frequency is also used to update the Decrementer (see 6.3, "Decrementer" on page 71), which is used by system software to set interval timers ("alarms"). The update frequency chosen should be appropriate for this purpose as well.

6.2.1 Writing and Reading the Time Base on 64-bit Implementations

Writing the Time Base is privileged; reading the Time Base is *not* privileged.

The 64-bit contents of a GPR may be written to the Time Base by the ***mtspr*** instruction. An extended mnemonic is provided which encodes the SPR number of the Time Base so that the number need not be specified as an operand; see page 75. To write the contents of register Rx to the Time Base, execute:

```
mttb  Rx
```

At the next Time Base update, the value written by ***mttb*** will be incremented by 1.

The contents of the Time Base may be read into a 64-bit GPR by the ***mftb*** instruction. An extended mnemonic (p. 75) is provided for this as well. To read the contents of the Time Base into register Rx, execute:

```
mftb  Rx
```

Reading the Time Base has no effect on the value it contains or the periodic incrementing of that value.

6.2.2 Writing and Reading the Time Base on 32-bit Implementations

Writing the Time Base is privileged; reading the Time Base is *not* privileged.

It is not possible to write or read the entire 64-bit Time Base in a single instruction on 32-bit machines. The ***mttb*** and ***mftb*** extended mnemonics move the lower half of the Time Base (TBL), while the ***mttbu*** and ***mftbu*** extended mnemonics move the upper half (TBU). These are extended mnemonics for the ***mtspr*** and ***mftb*** instructions; see page 75.

On a 32-bit implementation, ***mttb*** writes the contents of the specified GPR to TBL and writes zero to TBU.

The Time Base can be written by a sequence such as:

```
lwz    Rx,upper   # load 64-bit value for
lwz    Ry,lower   #   TB into Rx and Ry
li     Rz,0
mttb   Rz         # force TBL to 0
mttbu  Rx         # set TBU
mttb   Rz         # set TBL
```

Loading 0 into TBL prevents the possibility of a carry from TBL to TBU while the Time Base is being initialized.

Because of the possibility of a carry from TBL to TBU, a sequence such as the following is necessary to read the Time Base on 32-bit implementations.

```
loop:
   mftbu  Rx        # load from TBU
   mftb   Ry        # load from TBL
   mftbu  Rz        # load from TBU
   cmpw   Rz,Rx     # see if 'old' = 'new'
   bne    loop      # loop if carry occurred
```

The comparison and loop are necessary to ensure that a consistent pair of values have been obtained.

Programming Note

The ***mttbu*** and ***mftbu*** extended mnemonics are provided even on 64-bit implementations so that code written to read and write the Time Base on 32-bit implementations will work properly on both 32- and 64-bit implementations.

6.3 Decrementer

The Decrementer (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a Decrementer Interrupt after a programmable delay.

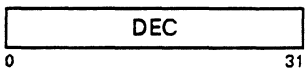


Figure 33. Decrementer

The Decrementer is driven by the same frequency as the Time Base. The period of the Decrementer will depend on the driving frequency, but if the same values are used as given above for the Time Base (section 6.2), and if the Time Base update frequency is constant, the period would be

$$T_{DEC} = \frac{2^{32} \times 32}{100\ \text{MHz}} = 1.37 \times 10^{3}\ \text{seconds}$$

which is approximately 23 minutes.

The Decrementer counts down, causing an interrupt (unless masked) when passing through zero. The Decrementer must be implemented such that the following requirements are satisfied:

1. The operation of the Time Base and the Decrementer are coherent, i.e. the counters are driven by the same fundamental time base.
2. Loading a GPR from the Decrementer shall have no effect on the Decrementer.
3. Storing a GPR to the Decrementer shall replace the value in the Decrementer with the value in the GPR.
4. Whenever bit 0 of the Decrementer changes from 0 to 1, an interrupt request is signalled. If multiple Decrementer Interrupt requests are received before the first can be reported, only one interrupt is reported. The occurrence of a Decrementer Interrupt cancels the request.
5. If the Decrementer is altered by software and the content of bit 0 is changed from 0 to 1, an interrupt request is signaled.

Programming Note

In systems that change the Time Base update frequency for purposes such as power management, the Decrementer input frequency will also change. Software must be aware of this in order to set interval timers.

On an implementation that performs speculative execution, the Decrementer may be read arbitrarily far "ahead" of the point at which it appears in the instruction stream. If it is important that this not occur, a context synchronizing operation such as the ***isync*** instruction should be placed immediately before the instruction that reads the Decrementer.

6.3.1 Writing and Reading the Decrementer

The content of the Decrementer can be read or written using the ***mfspr*** and ***mtspr*** instructions, both of which are privileged when they refer to the Decrementer. Using an extended mnemonic (see 75), the Decrementer may be written from register GPR Rx with:

```
mtdec  Rx
```

Programming Note

If the execution of this instruction causes bit 0 of the Decrementer to change from 0 to 1, an interrupt request is signalled.

The Decrementer may be read into GPR Rx with:

```
mfdec  Rx
```

Copying the Decrementer to a GPR has no effect on the Decrementer content or interrupt mechanism.

Appendix A. Optional Facilities and Instructions

The facilities (special purpose registers and instructions) described in this appendix are *optional*. An implementation may choose to provide all, some, or none of them. If a facility is implemented that matches semantics of a facility described here, the implementation should be as specified here.

A.1 External Control

The External Control facility provides a means for a problem state program to communicate with a special purpose device. Two instructions are provided:

- External Control Out Word Indexed (***ecowx***), which does the following:
  - Computes an Effective Address (EA) as for any X-form instruction
  - Validates the EA as would be done for a store to that address
  - Translates the EA to a Real Address
  - Transmits the Real Address and a word of data from a general register to the device
- External Control In Word Indexed (***eciwx***), which does the following:
  - Computes an Effective Address (EA) as for any X-form instruction
  - Validates the EA as would be done for a load from that address
  - Translates the EA to a Real Address
  - Transmits the Real Address to the device
  - Accepts a word of data from the device and places it in a general register

Depending on the setting of a control bit in a special purpose register, the External Access Register (EAR), the processor either performs the external control operation or it takes a Data Storage interrupt. The EAR controls access to the external access facility. Access to the EAR itself is privileged; the operating system can determine which tasks are allowed to issue External Access instructions and when they are allowed to do so.

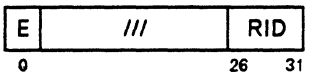
Interpretation of the real address transmitted by ***ecowx*** and ***eciwx*** and the 32-bit value transmitted by ***ecowx*** is up to the target device. Such interpretation is not specified by PowerPC Architecture. See the System Architecture documentation for a given PowerPC system for details on how the External Control facility can be used with devices on that system.

Example

An example of a device designed to be used with the External Control facility might be a graphics adapter. The ***ecowx*** instruction might be used to send the device the translated real address of a buffer containing graphics data, and the word transmitted from the general register might be control information that tells the adapter what operation to perform on the data in the buffer. The ***eciwx*** instruction might be used to load status information from the adapter.

A.1.1 External Access Register

This 32-bit Special Purpose Register controls access to the External Control facility and, for external control operations that are permitted, determines which device is the target.



Bit	*Name*	*Description*
0	E	Enable bit
26:31	RID	Resource ID

All other fields are reserved.

Figure 34. External Access Register

A.1.2 External Access Instructions

External Control In Word Indexed X-form

eciwx RT,RA,RB

31	RT	RA	RB	310	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
if EAR_E = 1 then
   raddr ← address translation of EA
   send load request for raddr to
      device identified by EAR_RID
   RT ← 32‖0 || word from device
else
   DSISR_11 ← 1
   generate Data Storage interrupt
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If $EAR_E = 1$, a load request for the real address corresponding to EA sent to the device identified by EAR_{RID}, bypassing the cache. $RT_{0:31}$ is set to 0. The word returned by the device is placed in $RT_{32:63(0:31)}$.

If $EAR_E = 0$, a Data Storage interrupt is taken, with bit 11 of DSISR set to 1.

The ***eciwx*** instruction is supported for Effective Addresses that reference ordinary (T=0) segments and for EAs mapped by Data BAT registers. The instruction is not supported and the results are boundedly undefined for EAs in direct-store (T=1) segments and for EAs generated when $MSR_{DR} = 0$ (real addresses).

The access caused by this instruction is treated as a load from the location addressed by EA with respect to protection and reference and change recording.

Special Registers Altered:

None

External Control Out Word Indexed X-form

ecowx RS,RA,RB

31	RS	RA	RB	438	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
if EAR_E = 1 then
   raddr ← address translation of EA
   send store request for raddr to
      device identified by EAR_RID
   send (RS_32:63(0:31)) to device
else
   DSISR_11 ← 1
   generate Data Storage interrupt
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If $EAR_E = 1$, a store request for the real address corresponding to EA and the contents of $RS_{32:63(0:31)}$ are sent to the device identified by EAR_{RID}, bypassing the cache.

If $EAR_E = 0$, a Data Storage interrupt is taken, with bit 11 of DSISR set to 1.

The ***ecowx*** instruction is supported for Effective Addresses that reference ordinary (T=0) segments and for EAs mapped by Data BAT registers. The instruction is not supported and the results are boundedly undefined for EAs in direct-store (T=1) segments and for EAs generated when $MSR_{DR} = 0$ (real addresses).

The access caused by this instruction is treated as a store to the location addressed by EA with respect to protection and reference and change recording.

Special Registers Altered:

None

Appendix B. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided that defines simple shorthand for the most frequently used forms of *Branch Conditional, Compare, Trap, Rotate and Shift,* and certain other instructions.

Most extended mnemonics are defined in an appendix to Book I, *PowerPC User Instruction Set Architecture*. Defined here are mnemonics related to ***mtspr*** and ***mfspr***, including privileged SPRs.

PowerPC-compliant assemblers will provide the mnemonics and symbols listed here and in the appendix cited above, and possibly others. Programs written to be portable across various assemblers for the PowerPC Architecture should not assume the existence of mnemonics not defined in the PowerPC Architecture books.

B.1 Move To/From Special Purpose Register mnemonics

The ***mtspr*** and ***mfspr*** instructions specify a Special Purpose Register (SPR) as a numeric operand. Extended mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as an operand. Also shown here are extended mnemonics for Move From Time Base and Move From Time Base Upper, which are variants of the ***mftb*** instruction rather than of ***mfspr***.

Note: ***mftb*** serves as both a basic and an extended mnemonic. The assembler will recognize an ***mftb*** mnemonic with two operands as the basic form, and an ***mftb*** mnemonic with one operand as the extended form.

Table 1. Extended mnemonics for moving to/from an SPR

Special Purpose Register	Move To SPR		Move From SPR[1]	
	Extended	*Equivalent to*	*Extended*	*Equivalent to*
Fixed Point Exception Register	mtxer Rx	mtspr 1,Rx	mfxer Rx	mfspr Rx,1
Link Register	mtlr Rx	mtspr 8,Rx	mflr Rx	mfspr Rx,8
Count Register	mtctr Rx	mtspr 9,Rx	mfctr Rx	mfspr Rx,9
Data Storage Interrupt Status Register	mtdsisr Rx	mtspr 18,Rx	mfdsisr Rx	mfspr Rx,18
Data Address Register	mtdar Rx	mtspr 19,Rx	mfdar Rx	mfspr Rx,19
Decrementer	mtdec Rx	mtspr 22,Rx	mfdec Rx	mfspr Rx,22
Storage Description Register 1	mtsdr1 Rx	mtspr 25,Rx	mfsdr1 Rx	mfspr Rx,25
Save/Restore Register 0	mtsrr0 Rx	mtspr 26,Rx	mfsrr0 Rx	mfspr Rx,26
Save/Restore Register 1	mtsrr1 Rx	mtspr 27,Rx	mfsrr1 Rx	mfspr Rx,27
Special Purpose Registers G0 through G3	mtsprg *n*,Rx	mtspr 272+*n*,Rx	mfsprg Rx,*n*	mfspr Rx,272+*n*
Address Space Register	mtasr Rx	mtspr 280,Rx	mfasr Rx	mfspr Rx,280
External Access Register	mtear Rx	mtspr 282,Rx	mfear Rx	mfspr Rx,282
Time Base [Lower]	mttb Rx	mtspr 284,Rx	mftb Rx	mftb Rx,268
Time Base Upper	mttbu Rx	mtspr 285,Rx	mftbu Rx	mftb Rx,269
Processor Version Register	–	–	mfpvr Rx	mfspr Rx,287
IBAT Registers, Upper	mtibatu *n*,Rx	mtspr 528+2×*n*,Rx	mfibatu Rx,*n*	mfspr Rx,528+2×*n*
IBAT Registers, Lower	mtibatl *n*,Rx	mtspr 529+2×*n*,Rx	mfibatl Rx,*n*	mfspr Rx,529+2×*n*
DBAT Registers, Upper	mtdbatu *n*,Rx	mtspr 536+2×*n*,Rx	mfdbatu Rx,*n*	mfspr Rx,536+2×*n*
DBAT Registers, Lower	mtdbatl *n*,Rx	mtspr 537+2×*n*,Rx	mfdbatl Rx,*n*	mfspr Rx,537+2×*n*

[1]Except for ***mftb*** and ***mftbu***.

Appendix C. Cross-Reference for Changed Power Mnemonics

The following table lists the Power instruction mnemonics that have been changed in the PowerPC Operating Environment Architecture, sorted by Power mnemonic.

To determine the PowerPC mnemonic for one of these Power mnemonics, find the Power mnemonic in the second column of the table: the remainder of the line gives the PowerPC mnemonic and the page on which the instruction is described, as well as the instruction names.

Power mnemonics that have not changed are not listed. Power instruction names that are the same in PowerPC are not repeated: i.e., for these, the last column of the table is blank.

Page	Power		PowerPC	
	Mnemonic	**Instruction**	**Mnemonic**	**Instruction**
46	mtsri	Move To Segment Register Indirect	mtsrin	
9	svca	Supervisor Call	sc	System Call
50	tlbi	TLB Invalidate Entry	tlbie	TLB Entry Invalidate

Appendix D. New Instructions

The following instructions in the PowerPC Operating Environment Architecture are new: they are not in the Power Architecture. ***dcbi*** and the Time Base instructions exist in all PowerPC implementations, ***mfsrin*** exists only in 32-bit implementations, and the SLB instructions exist only in 64-bit implementations. The SLB and TLB instructions are optional.

dcbi	Data Cache Block Invalidate
eciwx	External Control In Word Indexed
ecowx	External Control Out Word Indexed
mfsrin	Move From Segment Register Indirect
slbie	SLB Invalidate Entry
slbiex	SLB Invalidate Entry by Index
slbia	SLB Invalidate All
tlbiex	TLB Invalidate Entry by Index
tlbia	TLB Invalidate All
tlbsync	TLB Synchronize

Appendix E. Processor Version Numbers

The "processor version number" is the value contained in bits 0:15 of the Processor Version Register (PVR). This read-only register is described in section 2.2.4, "Processor Version Register" on page 8. The processor version number is uniquely determined by the specific version of the PowerPC Architecture implemented by a given processor. The value that a given processor should return is assigned by the PowerPC Architecture process.

Processor version numbers assigned as of 14 June 1992 are (hexadecimal):

0001
0003
0004
0014

Appendix F. Synchronization Requirements for Special Registers

The processor checks for input and output dependences with respect to all registers, and honors these dependences when executing a series of instructions involving a given register. For example, if an ***mtspr*** instruction writes a value to a particular SPR and an ***mfspr*** instruction later in the instruction stream reads the same SPR, the ***mfspr*** receives the value written by the ***mtspr***.

Such dependence checking does not extend to certain *side effects* of writing to status and control registers, SPRs, and Segment Registers, nor to the setting of certain SPRs by interrupts, as described in the remainder of this appendix.

The processor automatically provides all synchronization required for the GPRs, FPRs, CR, LR, CTR, XER, FPSCR, SRR 0, SRR 1, DAR, DSISR, SPRG0 through SPRG3, Time Base, and Decrementer, and for the EE and RI bits of the MSR, including side effects. These registers and MSR bits are not discussed further, in this appendix.

For the remainder of this appendix, words like "before," "after," "preceding," "following," etc., when referring to instruction sequence, are with respect to program order. (Program order is defined in Book II, *PowerPC Virtual Environment Architecture*.)

F.1 Affected Registers

Software synchronization may be required for alteration of the registers listed in the following subsections, because they affect instruction fetch and data access.

F.1.1 Instruction Fetch

Altering the content of the following registers or MSR bits may change the manner in which instruction addresses are interpreted, or the context in which instructions execute.

- ASR
- Segment Registers
- SDR 1
- IBAT registers
- MSR bits:
  PR, FP, ME, FE0, FE1, SE, BE, IP, IR, SF

F.1.2 Data Access

Altering the content of the following registers or MSR bits may change the manner in which data accesses are performed, or the context in which they are performed.

- ASR
- Segment Registers
- SDR 1
- DBAT registers
- EAR
- MSR bits:
  PR, DR, SF

F.2 Context Synchronizing Operations

The following instructions and events comprise the context synchronizing operations (see Section 1.7.1, "Context Synchronization" on page 3). They can be used to synchronize alteration of the registers listed above, as described below.

- ***isync***
- ***sc***
- ***rfi***
- any interrupt, other than System Reset and Machine Check

(As described in Chapter 5, "Interrupts" on page 57, System Reset and Machine Check *are* context synchronizing if they are recoverable.)

The ***sync*** instruction, although not context-synchronizing, can sometimes be used to provide the required synchronization, as described below.

F.3 Software Synchronization Requirements

To ensure that instructions appear to execute in program order (i.e., with the correct semantics and in the correct context), software must use synchronization instructions, as described below, when altering any of the registers and MSR bits listed in F.1, "Affected Registers."

Sometimes advantage can be taken of the fact that certain instructions that occur naturally in the program, such as the ***rfi*** at the end of an interrupt handler, provide the required synchronization.

Before Alteration

If the corresponding relocation is enabled (IR=1 for Section F.1.1, DR=1 for Section F.1.2), a context synchronizing operation or ***sync*** instruction must precede an alteration of any of the registers listed in Section F.1, with the exception of SDR 1 and the MSR.

If the corresponding relocation is enabled, a ***sync*** instruction must precede an alteration of SDR 1. The ***sync*** forces alterations of Reference and Change bits, due to instructions before the alteration of SDR 1, to be made in the correct context.

No explicit synchronization is required before software alters the MSR, because ***mtmsr*** is execution synchronizing (see Section 1.7.2, "Execution Synchronization" on page 4).

After Alteration

If the corresponding relocation is enabled (IR=1 for Section F.1.1, DR=1 for Section F.1.2), a context synchronizing operation must follow an alteration of any of the registers listed in Section F.1, with the exception of the MSR.

A context synchronizing operation must follow an alteration of any of the MSR bits listed in Sections F.1.1 and F.1.2, except MSR_{IP} if software does not care which value of this bit is used for non-recoverable System Reset and Machine Check interrupts.

Instructions fetched and/or executed after the alteration but before the context synchronizing operation may be fetched and/or executed in either the context that existed before the alteration or the context established by the alteration.

Multiple Alterations

When several of the registers listed in Section F.1 are altered with no intervening instructions that are affected by the alterations, no context synchronizing operations or ***sync*** instructions are required between the alterations.

Examples

- A single Segment Register is to be altered in isolation:

```
isync
mtsr  SRn,Rx
isync
```

- All the Segment Registers are to be reloaded upon task dispatch at the end of an interrupt.

```
mtsr  SR0,R...
mtsr  SR1,R...
...
mtsr  SR15,R...
rfi
```

  Because this instruction sequence reloads all Segment Registers, it must be executed with $MSR_{IR}=0$ and therefore no synchronization is required before the Segment Registers are loaded. (If the Segment Register that is being used for instruction fetch is not to be reloaded, the sequence can be executed with $MSR_{IR}=1$, and still no such synchronization is required.) The ***rfi*** provides the needed synchronization after the Segment Registers have been loaded, and before subsequent instructions are fetched and subsequent loads and stores executed.

F.4 Additional Software Requirements

This section describes additional software requirements with respect to instruction fetching and address translation. The results of failing to satisfy these requirements are undefined.

MSR_{IR}
MSR_{IR} should be altered only from code that is mapped virtual equals real.

ASR
If $MSR_{IR}=1$, alteration of the ASR is permitted only if the instructions in storage immediately following the ***mtspr*** that alters the ASR are identical in both the old and the new address space. Any resulting changes in storage protection or storage access mode are not guaranteed to take effect until a context synchronizing operation is executed.

Segment Registers

No fields in the Segment Register that is being used for instruction fetch should be altered, with the exception of the Key bits (K_s and K_p). Alteration of the Key bits is always permitted. Any resulting changes in storage protection are not guaranteed to take effect until a context synchronizing operation is executed.

SDR 1

SDR 1 should be altered only when $MSR_{IR} = 0$.

IBAT registers

No fields in the IBAT Register that is being used for instruction fetch should be altered, with the exception of the Valid (V) bit and the Key bits (K_s and K_p). Alteration of the V bit is permitted only if the instructions in storage immediately following the ***mtspr*** that alters the IBAT register are also mapped by the segmented address translation mechanism to the same address, or if the instructions are duplicated in the newly mapped space. Alteration of the Key bits is always permitted. Any resulting changes in storage protection or storage access mode are not guaranteed to take effect until a context synchronizing operation is executed.

To make an IBAT register valid in a manner such that the IBAT register then translates the current instruction stream, the following sequence should be used if fields in both the upper and lower IBAT registers are being altered.

1. The V bit in the IBAT register should be set to zero.
2. The other fields in the IBAT register should be initialized appropriately while the V bit remains zero.
3. The V bit should be set to one.
4. A context synchronizing operation should be executed.

If all altered fields are contained in either the upper or lower IBAT register, a single ***mtspr*** suffices (a synchronizing operation is not necessarily required).

Appendix G. Implementation-Specific SPRs

This appendix lists Special Purpose Register (SPR) numbers assigned by the PowerPC Architecture Review Process for implementation-specific uses. If a register shown here is present in a particular implementation, a detailed description will be found in Book IV, *PowerPC Implementation Features*.

The intent of this list is to ensure that if an SPR is needed for a particular function on more than one implementation, the same SPR number will be used.

Note that ordering of the bits shown in the table below matches the descriptions in *Move To/From Special Purpose Register* on pages 13 and 14. The two 5-bit halves of the SPR number are reversed from the order in which they appear in an assembled instruction.

	SPR	Register	Privi-
decimal	$spr_{5:9}$ $spr_{0:4}$	name	leged
1023	11111 11111	PIR	yes
1022	11111 11110	FPECR	yes

Processor ID Register (PIR)

This register holds a value that distinguishes this processor from others in a multiprocessor.

Floating-Point Exception Cause Register (FPECR)

This register identifies the reason a Floating-Point Exception occurred.

Appendix H. Interpretation of the DSISR as set by an Alignment Interrupt

For most causes of Alignment interrupt, the interrupt handler will emulate the interrupting instruction. To do this, it needs the following characteristics of the interrupting instruction:

Load or store
Length (half, word, or double)
String, multiple, or elementary
Fixed or float
Update or non-update
Byte reverse or not
Is it ***dcbz***?

The PowerPC Architecture provides this information implicitly, by setting opcode bits in the DSISR that identify the interrupting instruction type. It is not necessary for the interrupt handler to load the interrupting instruction from storage. The mapping is unique except for a few exceptions that are discussed below. The near-uniqueness depends upon the fact that many instructions cannot cause an Alignment interrupt, such as the fixed- and floating-point arithmetic instructions and the byte-width loads and stores.

See Section 5.5.6, "Alignment Interrupt" on page 63 for a description of how the opcode and extended opcode is mapped to a DSISR value for an X-, D-, or DS-form instruction that causes an Alignment interrupt.

The table on the next page shows the inverse mapping: how the DSISR bits identify the interrupting instruction. The following notes apply to this table.

(1) The instructions ***lwz*** and ***lwarx*** give the same DSISR bits (all zero). But if ***lwarx*** causes an alignment interrupt, it is an invalid form, so it need not be emulated in any precise way. It is adequate for the Alignment interrupt handler to simply emulate the instruction as if it were an ***lwz***. It is important that the emulator use the address in the DAR, rather than computing it from RA/RB/D, because ***lwz*** and ***lwarx*** are different formats.

    If opcode 0 ("Illegal or reserved") can cause an alignment interrupt, it will be indistinguishable from ***lwarx*** and ***lwz***.

(2) These are distinguished by DSISR bits 12:13, which are not shown in the table.

The Alignment interrupt handler will not be able to distinguish a floating-point load or store interrupting because it is misaligned, or because it addresses direct-store. But this does not matter; in either case it will be emulated by doing the operation with fixed-point instructions.

The interrupt handler has no need to distinguish between an X-form instruction and the corresponding D- or DS-form instruction, if one exists. Therefore two such instructions may report the same DSISR value (all 32 bits). For example, ***stw*** and ***stwx*** may both report either the DSISR value shown in the following table for ***stw***, or that shown for ***stwx***.

If DSISR 15:21 is:	then it is either X-form opcode:	or D/DS-form opcode:	so the instruction is:
00 0 0000	00000xxx00	x00000	lwarx, lwz, reserved (1)
00 0 0001	00010xxx00	x00010	ldarx
00 0 0010	00100xxx00	x00100	stw
00 0 0011	00110xxx00	x00110	-
00 0 0100	01000xxx00	x01000	lhz
00 0 0101	01010xxx00	x01010	lha
00 0 0110	01100xxx00	x01100	sth
00 0 0111	01110xxx00	x01110	lmw
00 0 1000	10000xxx00	x10000	lfs
00 0 1001	10010xxx00	x10010	lfd
00 0 1010	10100xxx00	x10100	stfs
00 0 1011	10110xxx00	x10110	stfd
00 0 1100	11000xxx00	x11000	-
00 0 1101	11010xxx00	x11010	ld, ldu, lwa (2)
00 0 1110	11100xxx00	x11100	-
00 0 1111	11110xxx00	x11110	std, stdu (2)
00 1 0000	00001xxx00	x00001	lwzu
00 1 0001	00011xxx00	x00011	-
00 1 0010	00101xxx00	x00101	stwu
00 1 0011	00111xxx00	x00111	-
00 1 0100	01001xxx00	x01001	lhzu
00 1 0101	01011xxx00	x01011	lhau
00 1 0110	01101xxx00	x01101	sthu
00 1 0111	01111xxx00	x01111	stmw
00 1 1000	10001xxx00	x10001	lfsu
00 1 1001	10011xxx00	x10011	lfdu
00 1 1010	10101xxx00	x10101	stfsu
00 1 1011	10111xxx00	x10111	stfdu
00 1 1100	11001xxx00	x11001	-
00 1 1101	11011xxx00	x11011	-
00 1 1110	11101xxx00	x11101	-
00 1 1111	11111xxx00	x11111	-
01 0 0000	00000xxx01		ldx
01 0 0001	00010xxx01		-
01 0 0010	00100xxx01		stdx
01 0 0011	00110xxx01		-
01 0 0100	01000xxx01		-
01 0 0101	01010xxx01		lwax
01 0 0110	01100xxx01		-
01 0 0111	01110xxx01		-
01 0 1000	10000xxx01		lswx
01 0 1001	10010xxx01		lswi
01 0 1010	10100xxx01		stswx
01 0 1011	10110xxx01		stswi
01 0 1100	11000xxx01		-
01 0 1101	11010xxx01		-
01 0 1110	11100xxx01		-
01 0 1111	11110xxx01		-
01 1 0000	00001xxx01		ldux
01 1 0001	00011xxx01		-
01 1 0010	00101xxx01		stdux
01 1 0011	00111xxx01		-
01 1 0100	01001xxx01		-
01 1 0101	01011xxx01		lwaux
01 1 0110	01101xxx01		-
01 1 0111	01111xxx01		-
01 1 1000	10001xxx01		-
01 1 1001	10011xxx01		-
01 1 1010	10101xxx01		-
01 1 1011	10111xxx01		-
01 1 1100	11001xxx01		-
01 1 1101	11011xxx01		-
01 1 1110	11101xxx01		-
01 1 1111	11111xxx01		-

If DSISR 15:21 is:	then it is either X-form opcode:	or D/DS-form opcode:	so the instruction is:
10 0 0000	00000xxx10		-
10 0 0001	00010xxx10		-
10 0 0010	00100xxx10		stwcx.
10 0 0011	00110xxx10		stdcx.
10 0 0100	01000xxx10		-
10 0 0101	01010xxx10		-
10 0 0110	01100xxx10		-
10 0 0111	01110xxx10		-
10 0 1000	10000xxx10		lwbrx
10 0 1001	10010xxx10		-
10 0 1010	10100xxx10		stwbrx
10 0 1011	10110xxx10		-
10 0 1100	11000xxx10		lhbrx
10 0 1101	11010xxx10		-
10 0 1110	11100xxx10		sthbrx
10 0 1111	11110xxx10		-
10 1 0000	00001xxx10		-
10 1 0001	00011xxx10		-
10 1 0010	00101xxx10		-
10 1 0011	00111xxx10		-
10 1 0100	01001xxx10		eciwx
10 1 0101	01011xxx10		-
10 1 0110	01101xxx10		ecowx
10 1 0111	01111xxx10		-
10 1 1000	10001xxx10		-
10 1 1001	10011xxx10		-
10 1 1010	10101xxx10		-
10 1 1011	10111xxx10		-
10 1 1100	11001xxx10		-
10 1 1101	11011xxx10		-
10 1 1110	11101xxx10		-
10 1 1111	11111xxx10		dcbz
11 0 0000	00000xxx11		lwzx
11 0 0001	00010xxx11		-
11 0 0010	00100xxx11		stwx
11 0 0011	00110xxx11		-
11 0 0100	01000xxx11		lhzx
11 0 0101	01010xxx11		lhax
11 0 0110	01100xxx11		sthx
11 0 0111	01110xxx11		-
11 0 1000	10000xxx11		lfsx
11 0 1001	10010xxx11		lfdx
11 0 1010	10100xxx11		stfsx
11 0 1011	10110xxx11		stfdx
11 0 1100	11000xxx11		-
11 0 1101	11010xxx11		-
11 0 1110	11100xxx11		-
11 0 1111	11110xxx11		stfiwx
11 1 0000	00001xxx11		lwzux
11 1 0001	00011xxx11		-
11 1 0010	00101xxx11		stwux
11 1 0011	00111xxx11		-
11 1 0100	01001xxx11		lhzux
11 1 0101	01011xxx11		lhaux
11 1 0110	01101xxx11		sthux
11 1 0111	01111xxx11		-
11 1 1000	10001xxx11		lfsux
11 1 1001	10011xxx11		lfdux
11 1 1010	10101xxx11		stfsux
11 1 1011	10111xxx11		stfdux
11 1 1100	11001xxx11		-
11 1 1101	11011xxx11		-
11 1 1110	11101xxx11		-
11 1 1111	11111xxx11		-

Appendix I. Processor Simplifications for Uniprocessor Designs

Microprocessor designs that will not be used in symmetric multiprocessor (SMP) systems may adopt optimizations to avoid cost and design effort implementing functions that will never be used. Further optimizations may be adopted if the design will never be used in conjunction with an L2 cache.

The following list identifies of the areas in which these optimizations can be made:

1. Receipt of TLB entry invalidate requests from other processors. Since the design will not be used in SMP systems, this function is not required.
2. Communication of ***sync*** to external mechanisms. The function provided by the ***sync*** instruction can be completed by the processor with no need to communicate with external mechanisms.
   A. Does the design of any storage controller require a notification that a ***sync*** is being executed?
   B. Does the design of any graphics subsystem require a notification that a ***sync*** is being executed?
   C. Does the design of any other I/O mechanism require a notification that a ***sync*** is being executed?
3. Communication of ***eieio*** to external mechanisms. The function provided by the ***eieio*** instruction can be completed by the processor with no need to communicate with external mechanisms. It is assumed that no L2 cache is used or its operation is totally transparent, and that all other mechanisms perform storage operations in the order that they are received.
4. Communication of cache management operations to external caches. It is assumed that no L2 cache is used or its operation is totally transparent. The function of these instructions can be completed in the processor with no need to communicate with external mechanisms.
5. Communication of TLB invalidates to external mechanisms. Graphics subsystem device drivers that use the move virtual storage instructions may require notification of a TLB invalidation.

Architecture Note

There is a pending proposal for these functions, so this requirement is dependent on the resolution of that proposal.

Appendix J. PowerPC Operating Environment Instruction Set

Form	Opcode		Mode Dep.[1]	Page	Mnemonic	Instruction
	Primary	Extend				
X	31	470		45	dcbi	Data Cache Block Invalidate
X	31	310		74	eciwx	External Control In Word Indexed
X	31	438		74	ecowx	External Control Out Word Indexed
X	31	83		15	mfmsr	Move From Machine State Register
XFX	31	339		14	mfspr	Move From Special Purpose Register
X	31	595	{}	46	mfsr	Move From Segment Register
X	31	659	{}	46	mfsrin	Move From Segment Register Indirect
X	31	146		15	mtmsr	Move To Machine State Register
XFX	31	467		13	mtspr	Move To Special Purpose Register
X	31	210	{}	46	mtsr	Move To Segment Register
X	31	242	{}	46	mtsrin	Move To Segment Register Indirect
XL	19	50		10	rfi	Return From Interrupt
SC	17	1		9	sc	System Call
X	31	498	()	49	slbia	SLB Invalidate All
X	31	434	()	48	slbie	SLB Invalidate Entry
X	31	466	()	48	slbiex	SLB Invalidate Entry by Index
X	31	370		52	tlbia	TLB Invalidate All
X	31	306		50	tlbie	TLB Invalidate Entry
X	31	338		51	tlbiex	TLB Invalidate Entry by Index
X	31	566		52	tlbsync	TLB Synchronize

[1]Key to Mode Dependency Column

Parentheses () are shown if the instruction is defined only for 64-bit implementations.

Braces {} are shown if the instruction is defined only for 32-bit implementations.

All instructions in the PowerPC Operating Environment Architecture are mode-independent, except that if the instruction refers to storage when in 32-bit mode, only the low-order 32 bits of the 64-bit effective address are used to address storage.

Index

R

S

T

U

V

W

Numerics

Last Page - End of Document

PowerPC Revisions

Revisions to: PowerPC Book 1, Rev. 0.05
Book 2, Rev. 0.04
Book 3, Rev. 0.03

These are changes for the most part agreed to, or in a few cases, under consideration, to the recently distributed books 1-3 of the PowerPC architecture. Most are documentation rather than functional issues.

The change notes are in some cases a little criptic, but they at least flag the areas being revised. Please contact Ron Hochsprung or John Sell with any comments.

John Sell (Sell.J, 4-5244), Ron Hochsprung (Hochsprung1, 4-2661)

Changes to Book 1, Revision 0.05, 4/14/92

We don't know of any open functional issues for book 1. The floating point exception mode performance guidance is still open as noted below; and there are a couple of new business items noted.
****

1.6.12.2 Agreed to change "invalid" class to "illegal" class. Invalid forms are still called invalid forms. So now there are defined, illegal and reserved instructions, and invalid forms of (defined) instructions.

1.6.13.2 In the programming note, using invalid forms "will result in" rather than "risk" incompatibility. Agreed to delete the programming note.

1.6.14 Would like to have the last paragraph to make the point that the normal mode for floating point exceptions is to be imprecise. Agreed to rewrite the paragraph. Also agreed that for now floating point exceptions are the only recoverable imprecise exceptions; in particular, page faults are defined to appear precise to software, but this may be revised as future business.

1.7.2 In the fourth paragraph, agreed to insert "appears to" into "effective address wraps around".

2.3.1 After the bullets, CR bits can't be tested in combination by branches. Agreed to revise wording.

2.4.1 Agreed to change the encoding of branch always so that the sign of the displacement and the "prediction bit" follow the same algorthm as conditional branches.

3.3.6 Agreed to delete the lswcbx instruction!!!

3.3.10 L=1 is not a mode that determines how operands are treated. It defines instruction encodings that aren't valid instructions in 32 bit mode or a 32 bit only implementations. Agreed to revise wording.

3.3.13.2 sradi's sub op field is 413 rather than 826.

4.1 Agreed to change "is not" to "may not be" in conformance (with IEEE) with NI set. However, NI mode won't be fully conforming to IEEE in forseeable implementations.

4.2.2 Agreed to add a new FPSCR bit which causes VX (invalid operation) to be set. This is to facilitate software implementation of IEEE conforming operations such as square root since VX cannot be set directly.

Agreed to make the "note" part of the regular text.

In the architectural note, agreed to delete the second paragraph. Also agreed to revise the architecture note so that it makes the following points. The purpose of NI mode is to always provide results with a guarenteed rate of execution; it will generally not significantly improve the overall performance of an application.

Agreed to delete the programming note.

4.3.5 In the second paragraph of item (3), agreed to delete the "must have at most 24 bits of significand".

Agreed to delete the engineering note.

4.4 The architecture has been changed, or clarified depending on one's view, so that any reading or writing to the floating point status register is content synchonizing. This means that the floating point status register always appears to software to reflect the sequencial state of the machine. Reading or writing it will also force any exceptions from preceding operations to happen first; sync is not required, and would typically have a negative impact on performance. The programming note will be revised accordingly.

Its been agreed to change the performance guidance section, following the programming note. There is still some disagreement as to what it will be. Following is our position, and our overview of the various floating point exception modes. (Cathy, the wording that Keith and I worked out at the meeting is fine; the following may not be exactly the same.)

For the best performance over the widest range of implementations, an application should use the imprecise, non recoverable mode if possible. Imprecise, recoverable mode should be used as a second choice. Precise mode is intended for diagnostic and specialized debugging purposes, and will be very slow in many implementations. Enabling the inexact exception will also result in much lower performance in many implementations. The FE01 = 00 exception disable mode provides compatibility with pre PowerPC implementations.

Floating Point Exception Modes

The non recoverable mode is, of course, not completely IEEE if any exceptions are enabled since the program cannot be continued with whatever the application wanted to do about the excepting operation. If all exceptions are disabled, this mode is fully IEEE; but future implementations should be using software assistance to deliver the proper result for many exception cases.

The precise and imprecise recoverable modes are fully IEEE, with software assistance used in future implementations to deliver the proper result for many exception cases. The exception models for these modes are as follows.

1. Each exception can be enabled individually. For example, inexact and underflow might be disabled since the disabled result is usually what the application would like, and invalid, divide by zero and overflow might be enabled so that the application can substitute its desired answer or take other special action.

2. For a disabled exception, the sticky bits or last operation bits can be tested at any time by the application.

3. For an enabled exception, the interrupt handler can take a default action, an application specified action or transfer to an application's handler.

We don't see how the override exceptions mode adds essential functionality to the above; however its certainly no problem having for compatibility with pre PowerPC. In this mode (FE01 = 00), all exceptions disabled or only inexact enabled yield the same results and capabilities as above. If one of the other exceptions is enabled, the results delivered (no result for invalid and divide by zero) are not suitable for continued processing without some special action. The application must test after each floating point operation. Going through the interrupt handler as in (3) above can be functionally equivalent. It can certainly be argued as to which is easier and (or) better performance. Going through the interrupt handler isn't high performance if the enabled exceptions happen often. But adding instructions after every operation to test the exception bits is worse in most situations. So in the interest of reducing the instances of multiple ways to do things and the attendent complexity and confusion, we would like to phase support for the FE01 = 00 mode out over time (like T = 1).

4.6 Agreed to delete the introductory paragraphs.

4.6.5 Agreed to investigate suggestions for potentially faster convert to unsigned integer examples; and to provide a complete set of 32 bit implementation only examples.

Changes to Book 2, Revision 0.04, 4/14/92

We don't know of any open functional issues for book 2 assuming that the aliasing issue is closed. However, as new business we are considering using OSA rather than cache inhibit as the means of preventing store operation gathering. It was also agreed that some of the wording revisions may be finished as new business rather than for revision 1.0.
****

1.2 Agreed to change the last part of the first paragraph to say "A simple model for sequential execution ..." rather than "A uni processor model".

1.3 Agreed to to change the definition of the page coherency attribute to be "required / not required" rather than disabled in this and following sections. Agreed to change the explanation on the next page to say that a processor is not required to perform coherency operations when the attribute is off rather than "inhibited from"; and "may not" rather than "will not".

Under caching inhibited, agreed to say "When caching is inhibited, the write through attribute has no meaning (period)." The rest of the sentence implies that one must do something that isn't required.

Under coherency, agreed to say "ensures that all copies of a storage location appear to be identical" rather than "are", as this is what really happens (one copy is valid and the others aren't actually identical).

1.3.1.2 "Coherency not Required" rather than "disabled"

Agreed to add a description of the architecturally required provision for other entities in the system to request coherency for a transaction.

All processors at the book 2 level will be able to make a transaction coming from another entity (to storage) coherent if told to as part of the transaction. This is different from making one's own transaction coherent because the page table said to. We want to say this here so that its understood that things like DMA I/O can be done to and from pages without setting the page table coherent bit. Otherwise most of the page table coherent bits would be set. In fact, no page coherency bits should need to be set in systems where only one entity has a cache. Leaving the page coherency bits off where possible minimizes bus traffic and improves performance.

1.4 Agreed to revise all of section 1.4xx as new business, including the

following points.

A typical situation will be to provide system services that programs call to do things like make data be executable as code, and most importantly that these service routines can be optimized to do the minimum necessary for the particular implementation.

The cache operations described (in book 2) are user operations; and that beyond the data to code issue, many of them are there for graphics and other programs to optimize the use of memory bandwidth where worthwhile.

Include a programming note to the effect that when the system knows that there is only one entity with caches (eg. a one general purpose processor system), all pages should have coherency "not required". This will reduce bus traffic and improve performance, especially in systems where there is a lot of graphics, video or other DMA.

1.4.1 The instructions are also defined to work appropriately with combined caches.

1.4.2 It would be more clear to just say that invalidate is a nop except that it broadcasts if coherency is required, and its OK to check the address.

1.5 Agreed that all implementations will be designed so that it is possible to support an aliasing on a page basis. In some cases this may require the addition of external logic. The following is OK as far as implementations go. My notes say that we did agree to say at the architectural level that aliasing is allowed on a page basis (period). Is this issue closed?

1.5.1.1 Agreed to consider allowing gathering of store operations for cache inhibited data as new business. This is currently not allowed. If this change were made, then OSA would be revised so that gathering was not allowed across an OSA.

1.5.1.2 The third bullet should say "completed" instead of "competed".

1.5.2.1 Agreed to clarify that reservation addresses must be specified as coherency required in the page table (in a multiple cache system) rather than being implied by the instructions.

2. Agreed to delete the seven numbered attributes section and the two sentences immediately preceding them.

2.2 Agreed to add the example of operations to I/O registers.

3.1 Agreed to delete the specific bit values left over from the deleted memory access parameters instruction.

3.2 Agreed to add that many of the operations may not be required to make code coherent in particular implementations; and say that its suggested that code be made coherent by calling a system service which does what is necessary for the particular implementation.

3.3 See aliasing issue at (1.5)

Agreed to add an architecture or programming note about the real difference between touch load and touch store; that is, the second one makes sure that the copy is exclusive.

3.4 In the engineering note, agreed to change "need not stop the processor" to "does not synchronize all operations in the processor".

4.1 Agreed to define a new 64 bit RTC which counts clock tics (typically divided by a small power of two). The old one will continue to exist for compatibility with pre PowerPC implementations; but there will be a note recommending that it not be used in new software, and that reading and writing it may be slow in future implementations. The new RTC will be read and written by new instruction encodings rather than new SPR numbers so that the 601 will trap.

Changes to Book 3, Revision 0.03, 4/13/92

We don't know of any open functional issues for book 3. There are a few new business items noted.
****

1.3 First bullet, PSR should be PMR.

1.3.1 Fourth bullet, a trap will refer to a taken trap and not the trap instruction itself.

There is confusion about the definition of "hardware". Agreed to revise the definition to something equivalent to the following. "Hardware" means any combination of hardware and software assistance used to implement the architecture. Software assistance may involve means, including instructions, which are implementation dependent; and it may operate soley as an extension of the hardware, appearing to be invisible to all normal software. In some cases, software assistance may use means which are part of the architecture. For example, floating point corner cases will typically invoke implementation dependent assisting software through architected interrupt means.

1.3.2 Agreed to delete that reserved fields are ignored by the hardware. They should be, of course; but its sufficient to say that software must make them zero. Then that they are ignored doesn't have to be verified by an implementation.

2.2.5 Agreed to clarify that version means each PowerPC implemetation. That is, each implementation has a unique number regardless of who designed it or what architecture revision it may be. Also applies to description in Appendix A.

2.3.1 Agreed to make system call context synchronizing, including floating point. Note (2) will be revised to delete the part about except floating point. It should also be made clear in the interrupt chapter that all interrupts are also context synchronizing.

4.2.1 Agreed to consider adding an architecture note as new business under direct store segment to the effect that: I/O may be memory mapped instead of using the direct store segments. Direct store segments are not preferred for new software development as it would be desirable to simplify the architecture by phasing them out over time.

4.2.2 The two paragraphs about stores towards the end could be made more clear by combining them to simply say that stores finish completely, or call the alignment handler having done nothing, or have stopped at a protection boundary.

Between this section and 5.6, it would help to state the architecture specifications that these book 3 exception rules are supporting; atomic, interrupted/no repeat, and restartable. The goal of these book 3 rules is to make it so that one alignment handler will work for a wide range of implementations.

Agreed to revise wording to clarify this section and reference book 2 architectural specifications.

4.2.5 First bullet under instruction prefetch, agreed to change "permitted except that when" to "permitted. Note that when".

Agreed to add an architectural note about machine check. Machine checks resulting solely from speculative execution will not be reported. Machine checks which a non speculatively executed program could not guarantee to avoid may be reported. Cache or internal data path parity errors are examples of errors that would be reported if implemented. An error resulting from a memory
operation to the primary location for the memory address (which could be a special device rather than main memory) would not be reported.

4.3 Agreed to delete the one control path in the drawing. It was meant to illustrate logical process flow.

Agreed to delete or correct the last paragraph, which contradicts the BAT precedence rule.

4.4.1.3 Decided to not combine the primary and secondary hash into one 16 entry search in order to be compatible with the 601 (to late to change). Also 32 bit version at 4.5.2.3.

In the large programming note:

Agreed to delete (2) and (3). Also at 4.4.2.3 and 4.5.2.3. (2) is an implementation detail, and (3) merely says that these accesses follow the same rules as all others.

Agreed to delete (8), which is superceded by the revised context synchronizing rules in Appendix B. Agreed to delete the engineering note immediately preceding the large programming note.

4.4.2.1 Second paragraph before the engineering note should say "at least one-half ..." to be consistent with similar paragraph in 32 bit section.

4.4.2.2 Engineering note should say 52 rather than 32 bit address.

4.6.1 Agreed to delete 601 note.

4.6.3 Agreed that instructions not supported for t=1 are invalid forms. They don't necessarily trap.

4.7.1 Agreed to revise the programming note so that it is more clear that the architecture allows normal page table entries to lie within a BAT; but since the BAT overrides, its not necessary for all of the equivalent regular pages to be entered into the page table.

Agreed that there will be four code and four data BAT's. All powers of two sizes from 128 KB through 256 MB will be provided in both 32 bit only and 64/32 implementations.

4.7.2 Agreed to delete the 601 note. The 601 BAT configuration will deviate from the architecture because its too late to fix; as described in the 601 Book 4. Revise this section to conform with changes outlined in 4.7.1.

4.8.1 See Book 2, 1.5.1.1 regarding store gathering.

Under memory coherent, agreed to mention that other entities can request that their transactions be made coherent, as in similar feature to be added to book 2. Also because of this, M can be off for everything in a system where only one entity has a cache. Finally, if coherency is off, then hardware "need not" rather than "should not" enforce data coherency.

4.11 Agreed to move the page table update examples so that they follow the definitions of the instructions used.

Agreed to note how much less is required for a uni processor system. For example, usually only a sync after everything is done. Will also provide examples for uni processor systems.

4.12.1 Agreed to move the section 4.12.3 on cache somewhere else. Then 4.12 deals only with virtual address caching, that is look aside buffer operations. There will be introductory paragraphs explaining that there are a sets of functions that an implementation must have; how these may vary depending on the
look aside capabilities and degree of multiple processor support of the implementation; and that the instructions described are suggested models for the operations which implementations should select from or propose desired alternatives. The goal is to encourage software to constrain where the operations are used so that it would be practical for them to be different on an implementation; and to encourage implementations to not be unecessarily different.

Not agreed, but its our position that the first point of the engineering note should be changed so that it is equivalent to the third point; that is, its implementation dependent whether software or hardware keeps multiple sets of segment registers consistent.

4.12.3 Machine attributes instruction has been deleted.

5.2.1, 5.2.2 Agreed to make t=1 store errors precise and fold into regular data storage interrupt.

5.3 Agreed that reading fp status bits is content synchronizing.

5.5 Agreed to consider as new business whether power on reset should be moved from Book 4 to Book 3.

Agreed to add an architected entry point for floating point corner case assistance code; and list the minimum information that must be delivered by hardware. The details of what is delivered and where are implementation dependent. The minimum information and requirements are to be able to determine the operation, the source operands, the result register, where to resume execution since the assisted (or excepting) operation may be imprecise, and be reported one at a time in proper order.

5.5.2 Agreed to delete "and some may do no error checking" from the engineering note.

5.5.3 Agreed that stwcx and stdcx won't get a DSI if the reservation has been lost (to avoid unimportant implementation dependencies).

Agreed that DSISR is set to 1 if the translation is not found in either HTEG or a BAT.

5.5.6 Agreed to delete the first paragraph of the engineering note, and to make the second paragraph a separate section which is not a subset of alignment.

5.5.7 Agreed to clarify wording of invalid instructions; implementations are also allowed to generate for any invalid forms.

5.6 Agreed that this section will be updated to be consistant with the rules that have been agreed to about exactly which instructions may be partially completed, where they stop, and which ones are restartable and which ones must be finished from where they stopped.

5.7.2 Agreed to delete the last two paragraphs (too many questions about the model example).

6.x Agreed to define a new 64 bit RTC which counts clock tics (typically divided by a small power of two). The old one will continue to exist for compatibility with pre PowerPC implementations; but there will be a note recommending that it not be used in new software, and that reading and writing it may be slow in future implementations. The new RTC will be read and written by new instruction encodings rather than new SPR numbers so that the 601 will trap.

Appendix B.

Agreed to completely revise this section. It will detail context synchonizing requirements for SPR, TLB and segment registers. A major goal is to eliminate the need for special synchronization operations in frequently occuring cases. In most cases, reading or writing an SPR will context sychronize on the affected data. For example, reading and writing floating point status will be context synchronizing.

General.

Agreed as new business to add an appendix summarizing the major PowerPC architecture options; 64 bit, and symmetric multiple processors (optional hardware support for look aside buffer and code cache coherency operations).

PowerPC Notes for the 601

Introduction

This document describes where the 601 chip differs from the official 1.02 version of the PowerPC Architecture. The differences came about because the design of the 601 had to be "frozen" relatively early in the process of the PowerPC Architecture definition. Hence, several changes to the architecture were made which were not able to be reflected in the 601 chip.

In some cases, the second version of the 601 chip (called DD2) will incorporate changes to make it more compatible with the official architecture. These instances will be noted below.

General Differences

The PowerPC Architecture defines a new set of mnemonics for many existing POWER instructions. However, we do not seem to have an assembler capable of accepting the "official" mnemonics. This issue must be dealt with by either getting an assembler which works! or by pre-processing source code to map the mnemonics and/or generate **.long** values for new PowerPC instructions which don't have an existing mnemonic (e.g., **LWARX.**).

Book I

The biggest change to Book I occurred with the addition of support for Little-Endian address mode; this change is described in Appendix D. The first version of the 601 (with which we are building EVT PDM's and Smurfs) does not have this mode. The second version, DD2, will add support for Little-Endian.

Book II

With the exception of the Time Base, the 601 implements Book II as described. Since it has a unified cache with sectored lines, there are differences between coherency size (which relates to how cache tags are kept) and block sizes for cache instructions (which relate to the burst-mode unit of data).

Chapter 3. Storage Control Instructions

3.1 Parameters Useful to Application Programs

The parameters described in section 3.1 for the 601 are:

1. Page Size:	4 KBytes
2. Coherence block size:	64 Bytes
3. Reservation block size:	64 Bytes
4. Split or Combined caches:	Combined cache
5,7: Total cache size:	32 KBytes
6,8: cache line size:	64 Bytes
9: **dcbt**, **dcbtst** block size:	32 Bytes
10: **icbi** block size:	32 Bytes
11: **dcbz**, **dcbst**, **dcbf**, **dcbi**:	32 Bytes
12,13: Combined cache associativity:	8-way
14: See below for Time Base differences	

Chapter 4. Time Base

The official PowerPC Architecture defines the Time Base Register to be a 64-bit value which is incremented at some "implementation-dependent" frequency. In fact, the frequency will (usually) be the "bus clock", which is a divisor of the processor's clock.

However, the 601 uses the old POWER definition where the upper 32-bit register is meant to be seconds and the lower 32-bit register is nano-seconds. The lower register "rolls-over" at 1,000,000,000, whereupon the upper register is incremented and the lower register goes to 0.

To implement this, the 601 has a separate pin which is meant to be driven at 7.8125 MHz (128 nsec. cycle time); this, in turn, increments bit-24 of the lower register. In all of our implementations, the clock is really driven at 7.8336 MHz, so that the Time Base appears to run slightly faster than it should.

Book III

The major difference of the 601 versus the PowerPC Architecture involves the Block Address Translation mechanism. In the original PowerPC, there were only 4 BAT registers; the current PowerPC has 4 BAT registers for each of Instruction and Data. In addition, the format of the registers changed.

3.4.1 Move To/From System Registers Instructions

The tables defining register values for use in mfspr, mtspr are incorrect for the 601. The 601 only has a total of 4 BATs, whose addresses correspond to those listed as **IBATnx** in Figures 9 & 10.

4.2.1 Storage Segments

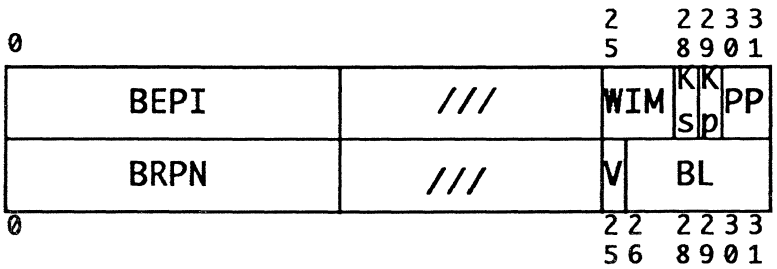
The 601 implements Direct Store Segments, using the T-bit in a Segment Register, as described in this section. However, unlike the official architecture, the 601 will always look at the T-bit, regardless of the state of the corresponding $\mathbf{MSR_{IR}}$ or $\mathbf{MSR_{DR}}$.

4.6 Direct-Store Segments

See above.

4.7 Block Address Translation

The description in Book III is generally correct, except for Figure 26. The correct version of Figure 26 for the 601 is:



The legal values for **BL** are:

000000	128 KB
000001	256 KB
000011	512 KB
000111	1 MB
001111	2 MB
011111	4 MB
111111	8 MB

4.8 Storage Access Modes

The 601 does not have a G-bit defined; i.e., Guarded Storage is not defined for the 601.

Chapter 6. Timer Facilities

This chapter needs to be interpreted in light of the discussion of Book II Time Base. The Decrementer register counts down at the "External Clock" rate, which is 7.8336 MHz for our systems.

Appendix A.1.2 External Access Instructions

While the 601 implements these, along with the EAR register, our systems will not work correctly if they are used. Assuming that the system never enables them (via the EAR), they will cause a Data Storage Interrupt.

```
**************************************************
*     +     new PowerPC instruction
*     -     old POWER instruction; not in PowerPC
*     P     Privileged instruction (Book III)
*     !     64-bit only instruction
*     ?     optional instruction
**************************************************

|0 0 0 0.0 0|? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?|   <<<<<undefined>>>>>
|0 0 0 0.0 1|? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?|   <<<<<undefined>>>>>
|0 0 0 0.1 0|   .T O  | .  R A  |       .    S I.1 6    .       |  ! TDI
|0 0 0 0.1 1|   .T O  | .  R A  |       .    S I.1 6    .       |    TWI            TI
|0 0 0 1.0 0|? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?|   <<<<<undefined>>>>>
|0 0 0 1.0 1|? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?|   <<<<<undefined>>>>>
|0 0 0 1.1 0|? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?|   <<<<<undefined>>>>>
|0 0 0 1.1 1|   .R T  | .  R A  |       .    S I.1 6    .       |    MULLI          MULI
|0 0 1 0.0 0|   .R T  | .  R A  |       .    S I.1 6    .       |    SUBFIC         SFI
|0 0 1 0.0 1|   .R T  | .  R A  |       .      D.1 6    .       |  -                DOZI
|0 0 1 0.1 0|  B.F|/|L| .  R A  |       .    S I.1 6    .       |    CMP[W|D]LI
|0 0 1 0.1 1|  B.F|/|L| .  R A  |       .    S I.1 6    .       |    CMP[W|D]I
|0 0 1 1.0 0|   .R T  | .  R A  |       .    S I.1 6    .       |    ADDIC          AI
|0 0 1 1.0 1|   .R T  | .  R A  |       .    S I.1 6    .       |    ADDIC.         AI.
|0 0 1 1.1 0|   .R T  | .  R A  |       .    S I.1 6    .       |    ADDI           CAL
|0 0 1 1.1 1|   .R T  | .  R A  |       .    S I.1 6    .       |    ADDIS          CAU
|0 1 0 0.0 0|   .B O  | .  B I  |       .  B D 1.4       .  |A|L|    BC[L][A]
|0 1 0 0.0 1|/ /./ / / /./ / / /./ / / /./ / / /./ / / /./ /|1|/|    SC             SVC[A]
|0 1 0 0.1 0|   .       .       .L I 2 4.       .       .  |A|L|    B[L][A]
|0 1 0 0.1 1|  B.F|/ /|B.F A|/ /|/ / / /./|0 0 0.0 0 0 0.0 0 0|/|    MCRF
|0 1 0 0.1 1|   .B O  | .  B I  |/ / / /./|0 0 0.0 0 1 0.0 0 0|L|    BCLR[L]       BCR[L]
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 0 0.0 1 0 0.0 0 1|/|    CRNOR
|0 1 0 0.1 1|/ /./ / /|/./ / / /|/ / / /./|0 0 0.0 1 1 0.0 1 0|/|  P  RFI
|0 1 0 0.1 1|/ /./ / /|/./ / / /|/ / / /./|0 0 0.1 0 1 0.0 1 0|/|  P-                RFSVC
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 0 1.0 0 0 0.0 0 1|/|    CRANDC
|0 1 0 0.1 1|/ /./ / /|/./ / / /|/ / / /./|0 0 1.0 0 1 0.1 1 0|/|    ISYNC          ICS
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 0 1.1 0 0 0.0 0 1|/|    CRXOR
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 0 1.1 1 0 0.0 0 1|/|    CRNAND
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 1 0.0 0 0 0.0 0 1|/|    CRAND
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 1 0.0 1 0 0.0 0 1|/|    CREQV
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 1 1.0 1 0 0.0 0 1|/|    CRORC
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 1 1.1 0 0 0.0 0 1|/|    CROR
|0 1 0 0.1 1|   .B O  | .  B I  |/ / / /./|1 0 0.0 0 1 0.0 0 0|L|    BCCTR[L]      BCC[L]
|0 1 0 1.0 0|   .R S  | .  R A  |    S H. |    m.b  |   .m e  |R|    RLWIMI[.]     RLIMI[.]
|0 1 0 1.0 1|   .R S  | .  R A  |    S H. |    m.b  |   .m e  |R|    RLWINM[.]     RLINM[.]
|0 1 0 1.1 0|   .R S  | .  R A  |    R B. |    m.b  |   .m e  |R|  -                RLMI[.]
|0 1 0 1.1 1|   .R S  | .  R A  |    R B. |    m.b  |   .m e  |R|    RLWNM[.]      RLNM[.]
|0 1 1 0.0 0|   .R S  | .  R A  |       .    U I.1 6    .       |    ORI            ORIL
|0 1 1 0.0 1|   .R S  | .  R A  |       .    U I.1 6    .       |    ORIS           ORIU
|0 1 1 0.1 0|   .R S  | .  R A  |       .    U I.1 6    .       |    XORI           XORIL
|0 1 1 0.1 1|   .R S  | .  R A  |       .    U I.1 6    .       |    XORIS          XORIU
|0 1 1 1.0 0|   .R S  | .  R A  |       .    U I.1 6    .       |    ANDI.          ANDIL.
|0 1 1 1.0 1|   .R S  | .  R A  |       .    U I.1 6    .       |    ANDIS.         ANDIU.
|0 1 1 1.1 0|   .R S  | .  R A  |    s h. |    m.b  |0 0.0 0|s|R|  !  RLDICL[.]
|0 1 1 1.1 0|   .R S  | .  R A  |    s h. |    m.e  |0 0.0 1|s|R|  !  RLDICR[.]
|0 1 1 1.1 0|   .R S  | .  R A  |    s h. |    m.b  |0 0.1 0|s|R|  !  RLDIC[.]
|0 1 1 1.1 0|   .R S  | .  R A  |    s h. |    m.b  |0 0.1 1|s|R|  !  RLDIMI[.]
|0 1 1 1.1 0|   .R S  | .  R A  |    R B. |    m.b  |0 1.0 0 0|R|  !  RLDCL[.]
|0 1 1 1.1 0|   .R S  | .  R A  |    R B. |    m.e  |0 1.0 0 1|R|  !  RLDCR[.]
|0 1 1 1.1 1|  B.F|/|L| .  R A  |    R B. |0 0 0.0 0 0 0.0 0 0|R|    CMP[W|D]
|0 1 1 1.1 1|   .T O  | .  R A  |    R B. |0 0 0.0 0 0 0.1 0 0|/|    TW             T
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|0 0.0 0 0 1.0 0 0|R|    SUBFC[O][.]   SF[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |/|0 0.0 0 0 1.0 0 1|R|  !  MULHDU[.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|0 0.0 0 0 1.0 1 0|R|    ADDC[O][.]    A[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |/|0 0.0 0 0 1.0 1 1|R|  +  MULHWU[.]
|0 1 1 1.1 1|   .R T  |/./ / / /|/ / / /./|0 0 0.0 0 1 0.0 1 1|/|    MFCR
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 0 0.0 0 1 0.1 0 0|/|  +  LWARX
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 0 0.0 0 1 0.1 0 1|/|  !  LDX
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 0 0.0 0 1 0.1 1 1|/|    LWZX           LX
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |0 0 0.0 0 1 1.0 0 0|R|    SLW[.]        SL[.]
|0 1 1 1.1 1|   .R S  | .  R A  |/ / / /./|0 0 0.0 0 1 1.0 1 0|R|    CNTLZW[.]     CNTLZ[.]
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |0 0 0.0 0 1 1.0 1 1|R|  !  SLD[.]
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |0 0 0.0 0 1 1.1 0 0|R|    AND[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|0 0 0.0 0 1 1.1 0 1|/|  -                MASKG[.]
|0 1 1 1.1 1|  B.F|/|L| .  R A  |    R B. |0 0 0.0 1 0 0.0 0 0|R|    CMP[W|D]L
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|0 0.0 1 0 1.0 0 0|R|  +  SUBF[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 0 0.0 1 1 0.1 0 1|/|  !  LDUX
|0 1 1 1.1 1|/ /./ / /|  .  R A  |    R B. |0 0 0.0 1 1 0.1 1 0|/|  +  DCBST
```

```
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |0 0 0.0 1 1 0.1 1 1|/|     LWZUX          LUX
|0 1 1 1.1 1|    .R S  | .  R A  |/ / / /./|0 0 0.0 1 1 1.0 1 0|R|   ! CNTLZD[.]
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 0.0 1 1 1.1 0 0|R|     ANDC[.]
|0 1 1 1.1 1|    .T O  | .  R A  |    R B. |0 0 0.1 0 0 0.1 0 0|/|   ! TD
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |/|0 0.1 0 0 1.0 0 1|R|   ! MULHD[.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |/|0 0.1 0 0 1.0 1 1|R|   + MULHW[.]
|0 1 1 1.1 1|    .R T  |/./ / / /|/ / / /./|0 0 0.1 0 1 0.0 1 1|/| P  MFMSR
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |0 0 0.1 0 1 0.1 0 0|/|   ! LDARX
|0 1 1 1.1 1|/ /./ / /| .  R A  |    R B. |0 0 0.1 0 1 0.1 1 0|/|   + DCBF
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |0 0 0.1 0 1 0.1 1 1|/|     LBZX
|0 1 1 1.1 1|    .R T  | .  R A  |/ / / /./|V|0 0.1 1 0 1.0 0 0|R|     NEG[O][.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|0 0.1 1 0 1.0 1 1|R|   -                MUL[O][.]
|0 1 1 1.1 1|/ /./ / /|./ / / /|/ / / /./|0 0 0.1 1 1 0.1 1 0|/|   -                CLF
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |0 0 0.1 1 1 0.1 1 1|/|     LBZUX
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 0.1 1 1 1.1 0 0|R|     NOR[.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|0 1.0 0 0 1.0 0 0|R|     SUBFE[O][.]    SFE[O][.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|0 1.0 0 0 1.0 1 0|R|     ADDE[O][.]     AE[O][.]
|0 1 1 1.1 1|    .R S  |/|     F.X M    |/|0 0 1.0 0 1 0.0 0 0|/|     MTCRF
|0 1 1 1.1 1|    .R S  |/./ / / /|/ / / /./|0 0 1.0 0 1 0.0 1 0|/| P  MTMSR
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 1.0 0 1 0.1 0 1|/|   ! STDX
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 1.0 0 1 0.1 1 0|1|   + STWCX.
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 1.0 0 1 0.1 1 1|/|     STWX           STX
|0 1 1 1.1 1|/ /./ / /|./ / / /|/ / / /./|0 0 1.0 0 1 1.0 0 0|/|   -                SLQ[.]
|0 1 1 1.1 1|/ /./ / /|./ / / /|/ / / /./|0 0 1.0 0 1 1.0 0 1|/|   -                SLE
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 1.0 1 1 0.1 0 1|/|   ! STDUX
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 1.0 1 1 0.1 1 1|/|     STWUX          STUX
|0 1 1 1.1 1|/ /./ / /|./ / / /|/ / / /./|0 0 1.0 1 1 1.0 0 0|/|   -                SLIQ[.]
|0 1 1 1.1 1|    .R T  | .  R A  |/ / / /./|V|0 1.1 0 0 1.0 0 0|R|     SUBFZE[O][.]  SFZE[O][.]
|0 1 1 1.1 1|    .R T  | .  R A  |/ / / /./|V|0 1.1 0 0 1.0 1 0|R|     ADDZE[O][.]   AZE[O][.]
|0 1 1 1.1 1|    .R S  |/|  S R  |/ / / /./|0 0 1.1 0 1 0.0 1 0|/| P  MTSR
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 1.1 0 1 0.1 1 0|1|   !  STDCX.
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 1.1 0 1 0.1 1 1|/|     STBX
|0 1 1 1.1 1|/ /./ / /|./ / / /|/ / / /./|0 0 1.1 0 1 1.0 0 0|/|   -                SLLQ[.]
|0 1 1 1.1 1|/ /./ / /|./ / / /|/ / / /./|0 0 1.1 0 1 1.0 0 1|/|   -                SLEQ[.]
|0 1 1 1.1 1|    .R T  | .  R A  |/ / / /./|V|0 1.1 1 0 1.0 0 0|R|     SUBFME[O][.]  SFME[O][.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|0 1.1 1 0 1.0 0 1|R|   !  MULLD[O][.]
|0 1 1 1.1 1|    .R T  | .  R A  |/ / / /./|V|0 1.1 1 0 1.0 1 0|R|     ADDME[O][.]   AME[O][.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|0 1.1 1 0 1.0 1 1|R|     MULLW[O][.]   MULS[O][.]
|0 1 1 1.1 1|    .R S  |/./ / / /|    R B. |0 0 1.1 1 1 0.0 1 0|/| P  MTSRIN         MTSRI
|0 1 1 1.1 1|/ /./ / /| .  R A  |    R B. |0 0 1.1 1 1 0.1 1 0|/|   +  DCBTST
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 1.1 1 1 0.1 1 1|/|     STBUX
|0 1 1 1.1 1|/ /./ / /|./ / / /|/ / / /./|0 0 1.1 1 1 1.0 0 0|/|   -                SLLIQ[.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|1 0.0 0 0 1.0 0 0|R|   -                DOZ[O][.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|1 0.0 0 0 1.0 1 0|R|     ADD[O][.]      CAX[O][.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |0 1 0.0 0 1 0.1 0 1|R|   -                LSCBX[.]
|0 1 1 1.1 1|/ /./ / /| .  R A  |    R B. |0 1 0.0 0 1 0.1 1 0|/|   +  DCBT
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |0 1 0.0 0 1 0.1 1 1|/|     LHZX
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 1 0.0 0 1 1.1 0 0|R|     EQV[.]
|0 1 1 1.1 1|/ /./ / /|./ / / /|    R B. |0 1 0.0 1 1 0.0 1 0|/| P? TLBIE          TLBI
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |0 1 0.0 1 1 0.1 1 0|/|   ?  ECIWX
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |0 1 0.0 1 1 0.1 1 1|/|     LHZUX
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 1 0.0 1 1 1.1 0 0|R|     XOR[.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|1 0.1 0 0 1.0 1 1|R|   -                DIV[O][.]
|0 1 1 1.1 1|/ /./ / /|./ / / /|    R B. |0 1 0.1 0 1 0.0 1 0|/| P? TLBIEX
|0 1 1 1.1 1|    .R T  | .     S.P R    . |0 1 0.1 0 1 0.0 1 1|/|     MFSPR
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |0 1 0.1 0 1 0.1 0 1|/|   !  LWAX
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |0 1 0.1 0 1 0.1 1 1|/|     LHAX
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|1 0.1 1 0 1.0 0 0|R|   -                ABS[O][.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|1 0.1 1 0 1.0 1 1|R|   -                DIVS[O][.]
|0 1 1 1.1 1|/ /./ / /|./ / / /|/ / / /./|0 1 0.1 1 1 0.0 1 0|/| P? TLBIA
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |0 1 0.1 1 1 0.1 0 1|/|   !  LWAUX
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |0 1 0.1 1 1 0.1 1 1|/|     LHAUX
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 1 1.0 0 1 0.1 1 1|/|     STHX
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 1 1.0 0 1 1.1 0 0|R|     ORC[.]
|0 1 1 1.1 1|/ /./ / /|./ / / /|    R B. |0 1 1.0 1 1 0.0 1 0|/| P! SLBIE
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 1 1.0 1 1 0.1 1 0|/|   ?  ECOWX
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 1 1.0 1 1 0.1 1 1|/|     STHUX
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 1 1.0 1 1 1.1 0 0|R|     OR[.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|1 1.1 0 0 1.0 0 1|R|   !  DIVDU[O][.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|1 1.1 0 0 1.0 1 1|R|   +  DIVWU[O][.]
|0 1 1 1.1 1|/ /./ / /|./ / / /|    R B. |0 1 1.1 0 1 0.0 1 0|/| P! SLBIEX
|0 1 1 1.1 1|    .R S  | .     S.P R    . |0 1 1.1 0 1 0.0 1 1|/|     MTSPR
|0 1 1 1.1 1|/ /./ / /| .  R A  |    R B. |0 1 1.1 0 1 0.1 1 0|/| P+ DCBI
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 1 1.1 0 1 1.1 0 0|R|     NAND[.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|1 1.1 1 0 1.0 0 0|R|   -                NABS[O][.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|1 1.1 1 0 1.0 0 1|R|   !  DIVD[O][.]
```

```
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|1 1.1 1 0 1.0 1 1|R|  + DIVW[O][.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|0 1 1.1 1 1 0.0 1 0|/| P! SLBIA
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|0 1 1.1 1 1 0.1 1 0|/| P-              CLI
|0 1 1 1.1 1|  B.F|/ /|/./ / / /|/ / / /./|1 0 0.0 0 0 0.0 0 0|/|    MCRXR
|0 1 1 1.1 1|   .R T  | .  R I  |/ / / /./|1 0 0.0 0 1 0.0 1 1|/|  -              CLCS
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |1 0 0.0 0 1 0.1 0 1|/|    LSWX          LSX
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |1 0 0.0 0 1 0.1 1 0|/|    LWBRX         LBRX
|0 1 1 1.1 1|  F.R T  | .  R A  |    R B. |1 0 0.0 0 1 0.1 1 1|/|    LFSX
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |1 0 0.0 0 1 1.0 0 0|R|    SRW[.]        SR[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 0.0 0 1 1.0 0 1|/|  -              RRIB[.]
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |1 0 0.0 0 1 1.0 1 1|R|  ! SRD[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 0.0 0 1 1.1 0 1|/|  -              MASKIR[.]
|0 1 1 1.1 1|  F.R T  | .  R A  |    R B. |1 0 0.0 1 1 0.1 1 1|/|    LFSUX
|0 1 1 1.1 1|   .R T  |/|  S R  |/ / / /./|1 0 0.1 0 1 0.0 1 1|/| P  MFSR
|0 1 1 1.1 1|   .R T  | .  R A  |    N B. |1 0 0.1 0 1 0.1 0 1|/|    LSWI          LSI
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 0.1 0 1 0.1 1 0|/|    SYNC          DCS
|0 1 1 1.1 1|  F.R T  | .  R A  |    R B. |1 0 0.1 0 1 0.1 1 1|/|    LFDX
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 0.1 1 1 0.0 1 1|/| P-              MFSRI
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 0.1 1 1 0.1 1 0|/|  -              DCLST
|0 1 1 1.1 1|  F.R T  | .  R A  |    R B. |1 0 0.1 1 1 0.1 1 1|/|    LFDUX
|0 1 1 1.1 1|   .R T  |/./ / / /|    R B. |1 0 1.0 0 1 0.0 1 1|/| P+ MFSRIN
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |1 0 1.0 0 1 0.1 0 1|R|    STSWX         STSX
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |1 0 1.0 0 1 0.1 1 0|/|    STWBRX        STBRX
|0 1 1 1.1 1|  F.R S  | .  R A  |    R B. |1 0 1.0 0 1 0.1 1 1|/|    STFSX
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 1.0 0 1 1.0 0 0|/|  -              SRQ[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 1.0 0 1 1.0 0 1|/|  -              SRE[.]
|0 1 1 1.1 1|  F.R S  | .  R A  |    R B. |1 0 1.0 1 1 0.1 1 1|/|    STFSUX
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 1.0 1 1 1.0 0 0|/|  -              SRIQ[.]
|0 1 1 1.1 1|   .R S  | .  R A  |    N B. |1 0 1.1 0 1 0.1 0 1|/|    STSWI         STSI
|0 1 1 1.1 1|  F.R S  | .  R A  |    R B. |1 0 1.1 0 1 0.1 1 1|/|    STFDX
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 1.1 0 1 1.0 0 0|/|  -              SRLQ[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 1.1 0 1 1.0 0 1|/|  -              SREQ[.]
|0 1 1 1.1 1|  F.R S  | .  R A  |    R B. |1 0 1.1 1 1 0.1 1 1|/|    STFDUX
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 1.1 1 1 1.0 0 0|/|  -              SRLIQ[.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |1 1 0.0 0 1 0.1 1 0|/|    LHBRX
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |1 1 0.0 0 1 1.0 0 0|R|    SRAW[.]       SRA[.]
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |1 1 0.0 0 1 1.0 1 0|R|  ! SRAD[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 1 0.0 1 1 0.0 1 0|/| P-              RAC
|0 1 1 1.1 1|   .R S  | .  R A  |    S H. |1 1 0.0 1 1 1.0 0 0|R|    SRAWI[.]      SRAI[.]
|0 1 1 1.1 1|   .R S  | .  R A  |    s h. |1 0 0.1 1 1 0.1 0|s|R|  ! SRADI[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 1 0.1 0 1 0.1 1 0|/|  + EIEIO
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |1 1 1.0 0 1 0.1 1 0|/|    STHBRX
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 1 1.0 0 1 1.0 0 0|/|  -              SRAQ[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 1 1.0 0 1 1.0 0 1|/|  -              SREA[.]
|0 1 1 1.1 1|   .R S  | .  R A  |/ / / /./|1 1 1.0 0 1 1.0 1 0|R|    EXTSH[.]      EXTS[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 1 1.0 1 1 1.0 0 0|/|  -              SRAIQ[.]
|0 1 1 1.1 1|   .R S  | .  R A  |/ / / /./|1 1 1.0 1 1 1.0 1 0|R|  + EXTSB[.]
|0 1 1 1.1 1|/ /./ / /| .  R A  |    R B. |1 1 1.1 0 1 0.1 1 0|/|  + ICBI
|0 1 1 1.1 1|  F.R S  | .  R A  |    R B. |1 1 1.1 0 1 0.1 1 1|/|  ? STFIWX
|0 1 1 1.1 1|   .R S  | .  R A  |/ / / /./|1 1 1.1 0 1 1.0 1 0|R|  ! EXTSW[.]
|0 1 1 1.1 1|/ /./ / /| .  R A  |    R B. |1 1 1.1 1 1 0.1 1 0|/|    DCBZ          DCLZ
|1 0 0 0.0 0|   .R T  | .  R A  |       .      D.1 6    .       |    LWZ           L
|1 0 0 0.0 1|   .R T  | .  R A  |       .      D.1 6    .       |    LWZU          LU
|1 0 0 0.1 0|   .R T  | .  R A  |       .      D.1 6    .       |    LBZ
|1 0 0 0.1 1|   .R T  | .  R A  |       .      D.1 6    .       |    LBZU
|1 0 0 1.0 0|   .R S  | .  R A  |       .      D.1 6    .       |    STW           ST
|1 0 0 1.0 1|   .R S  | .  R A  |       .      D.1 6    .       |    STWU          STU
|1 0 0 1.1 0|   .R S  | .  R A  |       .      D.1 6    .       |    STB
|1 0 0 1.1 1|   .R S  | .  R A  |       .      D.1 6    .       |    STBU
|1 0 1 0.0 0|   .R T  | .  R A  |       .      D.1 6    .       |    LHZ
|1 0 1 0.0 1|   .R T  | .  R A  |       .      D.1 6    .       |    LHZU
|1 0 1 0.1 0|   .R T  | .  R A  |       .      D.1 6    .       |    LHA
|1 0 1 0.1 1|   .R T  | .  R A  |       .      D.1 6    .       |    LHAU
|1 0 1 1.0 0|   .R S  | .  R A  |       .      D.1 6    .       |    STH
|1 0 1 1.0 1|   .R S  | .  R A  |       .      D.1 6    .       |    STHU
|1 0 1 1.1 0|   .R T  | .  R A  |       .      D.1 6    .       |    LMW           LM
|1 0 1 1.1 1|   .R S  | .  R A  |       .      D.1 6    .       |    STMW          STM
|1 1 0 0.0 0|  F.R T  | .  R A  |       .      D.1 6    .       |    LFS
|1 1 0 0.0 1|  F.R T  | .  R A  |       .      D.1 6    .       |    LFSU
|1 1 0 0.1 0|  F.R T  | .  R A  |       .      D.1 6    .       |    LFD
|1 1 0 0.1 1|  F.R T  | .  R A  |       .      D.1 6    .       |    LFDU
|1 1 0 1.0 0|  F.R S  | .  R A  |       .      D.1 6    .       |    STFS
|1 1 0 1.0 1|  F.R S  | .  R A  |       .      D.1 6    .       |    STFSU
|1 1 0 1.1 0|  F.R S  | .  R A  |       .      D.1 6    .       |    STFD
|1 1 0 1.1 1|  F.R S  | .  R A  |       .      D.1 6    .       |    STFDU
|1 1 1 0.0 0|? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?|   <<<<<undefined>>>>>
```

```
|1 1 1 0.0 1|? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?|   <<<<<undefined>>>>>
|1 1 1 0.1 0|   .R T  | .  R A  |       .  D S 1.4      .   |0 0|  ! LD
|1 1 1 0.1 0|   .R T  | .  R A  |       .  D S 1.4      .   |0 1|  ! LDU
|1 1 1 0.1 0|   .R T  | .  R A  |       .  D S 1.4      .   |1 0|  ! LWA
|1 1 1 0.1 1|  F.R T  | .F R A  |  F R B. |/ / /./ /|1 0.0 1 0|R|  + FDIVS[.]
|1 1 1 0.1 1|  F.R T  | .F R A  |  F R B. |/ / /./ /|1 0.1 0 0|R|  + FSUBS[.]
|1 1 1 0.1 1|  F.R T  | .F R A  |  F R B. |/ / /./ /|1 0.1 0 1|R|  + FADDS[.]
|1 1 1 0.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 0 1 0.1 1 0|R|  ? FSQRTS
|1 1 1 0.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 0 1 1.0 0 0|R|  ? FRES[.]
|1 1 1 0.1 1|  F.R T  | .F R A  |/ / / /./|  F R.C  |1 1.0 0 1|R|  + FMULS[.]
|1 1 1 0.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 0 1 1.0 1 0|R|  ? FRSQRTE[.]
|1 1 1 0.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 0 0|R|  + FMSUBS[.]
|1 1 1 0.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 0 1|R|  + FMADDS[.]
|1 1 1 0.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 1 0|R|  + FNMSUBS[.]
|1 1 1 0.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 1 1|R|  + FNMADDS[.]
|1 1 1 1.0 0|? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?|   <<<<<undefined>>>>>
|1 1 1 1.0 1|? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?.? ? ? ?|   <<<<<undefined>>>>>
|1 1 1 1.1 0|   .R S  | .  R A  |       .  D S 1.4      .   |0 0|  ! STD
|1 1 1 1.1 0|   .R S  | .  R A  |       .  D S 1.4      .   |0 1|  ! STDU
|1 1 1 1.1 1|  B.F|/ /| .F R A  |  F R B. |0 0 0.0 0 0 0.0 0 0|R|    FCMPU
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 0 0 1.1 0 0|R|    FRSP[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 0 0 1.1 1 0|R|  + FCTIW[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 0 0 1.1 1 1|R|  + FCTIWZ[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |/ / /./ /|1 0.0 1 0|R|    FDIV[.]       FD[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |/ / /./ /|1 0.1 0 0|R|    FSUB[.]       FS[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |/ / /./ /|1 0.1 0 1|R|    FADD[.]       FA[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 0 1 0.1 1 0|R|  ? FSQRT
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 0.1 1 1|R|  ? FSEL[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |/ / / /./|  F R.C  |1 1.0 0 1|R|    FMUL[.]       FM[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 0 0|R|    FMSUB[.]      FMS[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 0 1|R|    FMADD[.]      FMA[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 1 0|R|    FNMSUB[.]     FNMS[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 1 1|R|    FNMADD[.]     FNMA[.]
|1 1 1 1.1 1|  B.F|/ /| .F R A  |  F R B. |0 0 0.0 1 0 0.0 0 0|R|    FCMPO
|1 1 1 1.1 1|   .B T  |/./ / / /|/ / / /./|0 0 0.0 1 0 0.1 1 0|R|    MTFSB1[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 1 0 1.0 0 0|R|    FNEG[.]
|1 1 1 1.1 1|  B.F|/ /|B.F A|/ /|/ / / /./|0 0 0.1 0 0 0.0 0 0|/|    MCRFS
|1 1 1 1.1 1|   .B T  |/./ / / /|/ / / /./|0 0 0.1 0 0 0.1 1 0|R|    MTFSB0[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.1 0 0 1.0 0 0|R|    FMR[.]
|1 1 1 1.1 1|  B.F|/ /|/./ / / /|    U  |/|0 0 1.0 0 0 0.1 1 0|R|    MTFSFI[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 0 1.0 0 0 1.0 0 0|R|    FNABS[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 1 0.0 0 0 1.0 0 0|R|    FABS[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|/ / / /./|1 0 0.1 0 0 0.1 1 1|R|    MFFS[.]
|1 1 1 1.1 1|/| .    F L.M    |/|  F R B. |1 0 1.1 0 0 0.1 1 1|R|    MTFSF[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |1 1 0.0 1 0 1.1 1 0|R|  ! FCTID[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |1 1 0.0 1 0 1.1 1 1|R|  ! FCTIDZ[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |1 1 0.1 0 0 1.1 1 0|R|  ! FCFID[.]
```

```
**************************************************
*     +     new PowerPC instruction
*     -     old POWER instruction; not in PowerPC
*     P     Privileged instruction (Book III)
*     !     64-bit only instruction
*     ?     optional instruction
**************************************************

|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|1 0.1 1 0 1.0 0 0|R|  -                 ABS[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|1 0.0 0 0 1.0 1 0|R|    ADD[O][.]      CAX[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|0 0.0 0 0 1.0 1 0|R|    ADDC[O][.]     A[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|0 1.0 0 0 1.0 1 0|R|    ADDE[O][.]     AE[O][.]
|0 0 1 1.1 0|   .R T  | .  R A  |       .    S I.1 6    .       |    ADDI           CAL
|0 0 1 1.0 0|   .R T  | .  R A  |       .    S I.1 6    .       |    ADDIC          AI
|0 0 1 1.0 1|   .R T  | .  R A  |       .    S I.1 6    .       |    ADDIC.         AI.
|0 0 1 1.1 1|   .R T  | .  R A  |       .    S I.1 6    .       |    ADDIS          CAU
|0 1 1 1.1 1|   .R T  | .  R A  |/ / / /./|V|0 1.1 1 0 1.0 1 0|R|    ADDME[O][.]   AME[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |/ / / /./|V|0 1.1 0 0 1.0 1 0|R|    ADDZE[O][.]   AZE[O][.]
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |0 0 0.0 0 1 1.1 0 0|R|    AND[.]
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |0 0 0.0 1 1 1.1 0 0|R|    ANDC[.]
|0 1 1 1.0 0|   .R S  | .  R A  |       .    U I.1 6    .       |    ANDI.          ANDIL.
|0 1 1 1.0 1|   .R S  | .  R A  |       .    U I.1 6    .       |    ANDIS.         ANDIU.
|0 1 0 0.1 0|   .     .       .L I 2 4.       .       .   |A|L|    B[L][A]
|0 1 0 0.0 0|   .B O  | .  B I  |       .  B D 1.4        .   |A|L|    BC[L][A]
|0 1 0 0.1 1|   .B O  | .  B I  |/ / / /./|1 0 0.0 0 1 0.0 0 0|L|    BCCTR[L]      BCC[L]
|0 1 0 0.1 1|   .B O  | .  B I  |/ / / /./|0 0 0.0 0 1 0.0 0 0|L|    BCLR[L]       BCR[L]
|0 1 1 1.1 1|   .R T  | .  R I  |/ / / /./|1 0 0.0 0 1 0.0 1 1|/|  -                 CLCS
|0 1 1 1.1 1|/ /./ / /|./ / / /|/ / / /./|0 0 0.1 1 1 0.1 1 0|/|  -                 CLF
|0 1 1 1.1 1|/ /./ / /|./ / / /|/ / / /./|0 1 1.1 1 1 0.1 1 0|/| P-                 CLI
|0 0 1 0.1 1|  B.F|/|L| .  R A  |       .    S I.1 6    .       |    CMP[W|D]I
|0 1 1 1.1 1|  B.F|/|L| .  R A  |    R B. |0 0 0.0 0 0 0.0 0 0|R|    CMP[W|D]
|0 0 1 0.1 0|  B.F|/|L| .  R A  |       .    S I.1 6    .       |    CMP[W|D]LI
|0 1 1 1.1 1|  B.F|/|L| .  R A  |    R B. |0 0 0.0 1 0 0.0 0 0|R|    CMP[W|D]L
|0 1 1 1.1 1|   .R S  | .  R A  |/ / / /./|0 0 0.0 1 1 1.0 1 0|R|  !  CNTLZD[.]
|0 1 1 1.1 1|   .R S  | .  R A  |/ / / /./|0 0 0.0 0 1 1.0 1 0|R|    CNTLZW[.]     CNTLZ[.]
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 1 0.0 0 0 0.0 0 1|/|    CRAND
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 0 1.0 0 0 0.0 0 1|/|    CRANDC
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 1 0.0 1 0 0.0 0 1|/|    CREQV
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 0 1.1 1 0 0.0 0 1|/|    CRNAND
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 0 0.0 1 0 0.0 0 1|/|    CRNOR
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 1 1.1 0 0 0.0 0 1|/|    CROR
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 1 1.0 1 0 0.0 0 1|/|    CRORC
|0 1 0 0.1 1|   .B T  | .  B A  |    B B. |0 0 1.1 0 0 0.0 0 1|/|    CRXOR
|0 1 1 1.1 1|/ /./ / /| .  R A  |    R B. |0 0 0.1 0 1 0.1 1 0|/|  +  DCBF
|0 1 1 1.1 1|/ /./ / /| .  R A  |    R B. |0 1 1.1 0 1 0.1 1 0|/| P+  DCBI
|0 1 1 1.1 1|/ /./ / /| .  R A  |    R B. |0 0 0.0 1 1 0.1 1 0|/|  +  DCBST
|0 1 1 1.1 1|/ /./ / /| .  R A  |    R B. |0 1 0.0 0 1 0.1 1 0|/|  +  DCBT
|0 1 1 1.1 1|/ /./ / /| .  R A  |    R B. |0 0 1.1 1 1 0.1 1 0|/|  +  DCBTST
|0 1 1 1.1 1|/ /./ / /| .  R A  |    R B. |1 1 1.1 1 1 0.1 1 0|/|     DCBZ         DCLZ
|0 1 1 1.1 1|/ /./ / /|./ / / /|/ / / /./|1 0 0.1 1 1 0.1 1 0|/|  -                 DCLST
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|1 0.1 0 0 1.0 1 1|R|  -                 DIV[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|1 1.1 1 0 1.0 0 1|R|  !  DIVD[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|1 1.1 0 0 1.0 0 1|R|  !  DIVDU[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|1 0.1 1 0 1.0 1 1|R|  -                 DIVS[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|1 1.1 1 0 1.0 1 1|R|  +  DIVW[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|1 1.1 0 0 1.0 1 1|R|  +  DIVWU[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|1 0.0 0 0 1.0 0 0|R|  -                 DOZ[O][.]
|0 0 1 0.0 1|   .R T  | .  R A  |       .      D.1 6    .       |  -                 DOZI
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 1 0.0 1 1 0.1 1 0|/|  ?  ECIWX
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |0 1 1.0 1 1 0.1 1 0|/|  ?  ECOWX
|0 1 1 1.1 1|/ /./ / /|./ / / /|/ / / /./|1 1 0.1 0 1 0.1 1 0|/|  +  EIEIO
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |0 1 0.0 0 1 1.1 0 0|R|     EQV[.]
|0 1 1 1.1 1|   .R S  | .  R A  |/ / / /./|1 1 1.0 1 1 1.0 1 0|R|  +  EXTSB[.]
|0 1 1 1.1 1|   .R S  | .  R A  |/ / / /./|1 1 1.0 0 1 1.0 1 0|R|     EXTSH[.]     EXTS[.]
|0 1 1 1.1 1|   .R S  | .  R A  |/ / / /./|1 1 1.1 0 1 1.0 1 0|R|  !  EXTSW[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 1 0.0 0 0 1.0 0 0|R|     FABS[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |/ / /./ /|1 0.1 0 1|R|     FADD[.]      FA[.]
|1 1 1 0.1 1|  F.R T  | .F R A  |  F R B. |/ / /./ /|1 0.1 0 1|R|  +  FADDS[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |1 1 0.1 0 0 1.1 1 0|R|  !  FCFID[.]
|1 1 1 1.1 1|  B.F|/ /| .F R A  |  F R B. |0 0 0.0 1 0 0.0 0 0|R|     FCMPO
|1 1 1 1.1 1|  B.F|/ /| .F R A  |  F R B. |0 0 0.0 0 0 0.0 0 0|R|     FCMPU
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |1 1 0.0 1 0 1.1 1 0|R|  !  FCTID[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |1 1 0.0 1 0 1.1 1 1|R|  !  FCTIDZ[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 0 0 1.1 1 0|R|  +  FCTIW[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 0 0 1.1 1 1|R|  +  FCTIWZ[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |/ / /./ /|1 0.0 1 0|R|     FDIV[.]      FD[.]
```

```
|1 1 1 0.1 1|  F.R T  | .F R A  |  F R B. |/ / /./ /|1 0.0 1 0|R|  + FDIVS[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 0 1|R|    FMADD[.]      FMA[.]
|1 1 1 0.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 0 1|R|  + FMADDS[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.1 0 0 1.0 0 0|R|    FMR[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 0 0|R|    FMSUB[.]      FMS[.]
|1 1 1 0.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 0 0|R|  + FMSUBS[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |/ / / /./|  F R.C  |1 1.0 0 1|R|    FMUL[.]       FM[.]
|1 1 1 0.1 1|  F.R T  | .F R A  |/ / / /./|  F R.C  |1 1.0 0 1|R|  + FMULS[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 0 1.0 0 0 1.0 0 0|R|    FNABS[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 1 0 1.0 0 0|R|    FNEG[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 1 1|R|    FNMADD[.]     FNMA[.]
|1 1 1 0.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 1 1|R|  + FNMADDS[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 1 0|R|    FNMSUB[.]     FNMS[.]
|1 1 1 0.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 1.1 1 0|R|  + FNMSUBS[.]
|1 1 1 0.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 0 1 1.0 0 0|R|  ? FRES[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 0 0 1.1 0 0|R|    FRSP[.]
|1 1 1 0.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 0 1 1.0 1 0|R|  ? FRSQRTE[.]
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 0.1 1 1|R|  ? FSEL[.]
|1 1 1 1.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 0 1 0.1 1 0|R|  ? FSQRT
|1 1 1 0.1 1|  F.R T  |/./ / / /|  F R B. |0 0 0.0 0 1 0.1 1 0|R|  ? FSQRTS
|1 1 1 1.1 1|  F.R T  | .F R A  |  F R B. |  F R.C  |1 0.1 0 0|R|    FSUB[.]       FS[.]
|1 1 1 0.1 1|  F.R T  | .F R A  |  F R B. |/ / /./ /|1 0.1 0 0|R|  + FSUBS[.]
|0 1 1 1.1 1|/ /./ / /|  .  R A  |    R B. |1 1 1.1 0 1 0.1 1 0|/|  + ICBI
|0 1 0 0.1 1|/ /./ / /|./ / / /|/ / / /./|0 0 1.0 0 1 0.1 1 0|/|    ISYNC        ICS
|1 0 0 0.1 0|   .R T  | .  R A  |       .      D.1 6    .       |    LBZ
|1 0 0 0.1 1|   .R T  | .  R A  |       .      D.1 6    .       |    LBZU
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 0 0.1 1 1 0.1 1 1|/|    LBZUX
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 0 0.1 0 1 0.1 1 1|/|    LBZX
|1 1 1 0.1 0|   .R T  | .  R A  |       .  D S 1.4      .   |0 0|  ! LD
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 0 0.1 0 1 0.1 0 0|/|  ! LDARX
|1 1 1 0.1 0|   .R T  | .  R A  |       .  D S 1.4      .   |0 1|  ! LDU
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 0 0.0 1 1 0.1 0 1|/|  ! LDUX
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 0 0.0 0 1 0.1 0 1|/|  ! LDX
|1 1 0 0.1 0|  F.R T  | .  R A  |       .      D.1 6    .       |    LFD
|1 1 0 0.1 1|  F.R T  | .  R A  |       .      D.1 6    .       |    LFDU
|0 1 1 1.1 1|  F.R T  | .  R A  |    R B. |1 0 0.1 1 1 0.1 1 1|/|    LFDUX
|0 1 1 1.1 1|  F.R T  | .  R A  |    R B. |1 0 0.1 0 1 0.1 1 1|/|    LFDX
|1 1 0 0.0 0|  F.R T  | .  R A  |       .      D.1 6    .       |    LFS
|1 1 0 0.0 1|  F.R T  | .  R A  |       .      D.1 6    .       |    LFSU
|0 1 1 1.1 1|  F.R T  | .  R A  |    R B. |1 0 0.0 1 1 0.1 1 1|/|    LFSUX
|0 1 1 1.1 1|  F.R T  | .  R A  |    R B. |1 0 0.0 0 1 0.1 1 1|/|    LFSX
|1 0 1 0.1 0|   .R T  | .  R A  |       .      D.1 6    .       |    LHA
|1 0 1 0.1 1|   .R T  | .  R A  |       .      D.1 6    .       |    LHAU
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 1 0.1 1 1 0.1 1 1|/|    LHAUX
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 1 0.1 0 1 0.1 1 1|/|    LHAX
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |1 1 0.0 0 1 0.1 1 0|/|    LHBRX
|1 0 1 0.0 0|   .R T  | .  R A  |       .      D.1 6    .       |    LHZ
|1 0 1 0.0 1|   .R T  | .  R A  |       .      D.1 6    .       |    LHZU
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 1 0.0 1 1 0.1 1 1|/|    LHZUX
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 1 0.0 0 1 0.1 1 1|/|    LHZX
|1 0 1 1.1 0|   .R T  | .  R A  |       .      D.1 6    .       |    LMW           LM
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 1 0.0 0 1 0.1 0 1|R|  -             LSCBX[.]
|0 1 1 1.1 1|   .R T  | .  R A  |    N B. |1 0 0.1 0 1 0.1 0 1|/|    LSWI         LSI
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |1 0 0.0 0 1 0.1 0 1|/|    LSWX         LSX
|1 1 1 0.1 0|   .R T  | .  R A  |       .  D S 1.4      .   |1 0|  ! LWA
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 0 0.0 0 1 0.1 0 0|/|  + LWARX
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 1 0.1 1 1 0.1 0 1|/|  ! LWAUX
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 1 0.1 0 1 0.1 0 1|/|  ! LWAX
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |1 0 0.0 0 1 0.1 1 0|/|    LWBRX        LBRX
|1 0 0 0.0 0|   .R T  | .  R A  |       .      D.1 6    .       |    LWZ           L
|1 0 0 0.0 1|   .R T  | .  R A  |       .      D.1 6    .       |    LWZU          LU
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 0 0.0 1 1 0.1 1 1|/|    LWZUX        LUX
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |0 0 0.0 0 1 0.1 1 1|/|    LWZX         LX
|0 1 1 1.1 1|/ /./ / /|./ / / /|/ / / /./|0 0 0.0 0 1 1.1 0 1|/|  -             MASKG[.]
|0 1 1 1.1 1|/ /./ / /|./ / / /|/ / / /./|1 0 0.0 0 1 1.1 0 1|/|  -             MASKIR[.]
|0 1 0 0.1 1|  B.F|/ /|B.F A|/ /|/ / / /./|0 0 0.0 0 0 0.0 0 0|/|    MCRF
|1 1 1 1.1 1|  B.F|/ /|B.F A|/ /|/ / / /./|0 0 0.1 0 0 0.0 0 0|/|    MCRFS
|0 1 1 1.1 1|  B.F|/ /|./ / / /|/ / / /./|1 0 0.0 0 0 0.0 0 0|/|    MCRXR
|0 1 1 1.1 1|   .R T  |/./ / / /|/ / / /./|0 0 0.0 0 1 0.0 1 1|/|    MFCR
|1 1 1 1.1 1|  F.R T  |/./ / / /|/ / / /./|1 0 0.1 0 0 0.1 1 1|R|    MFFS[.]
|0 1 1 1.1 1|   .R T  |/./ / / /|/ / / /./|0 0 0.1 0 1 0.0 1 1|/|  P  MFMSR
|0 1 1 1.1 1|   .R T  | .      S.P R    . |0 1 0.1 0 1 0.0 1 1|/|    MFSPR
|0 1 1 1.1 1|   .R T  |/|  S R  |/ / / /./|1 0 0.1 0 1 0.0 1 1|/|  P  MFSR
|0 1 1 1.1 1|/ /./ / /|./ / / /|/ / / /./|1 0 0.1 1 1 0.0 1 1|/|  P-            MFSRI
|0 1 1 1.1 1|   .R T  |/./ / / /|    R B. |1 0 1.0 0 1 0.0 1 1|/|  P+ MFSRIN
|0 1 1 1.1 1|   .R S  |/|      F.X M    |/|0 0 1.0 0 1 0.0 0 0|/|    MTCRF
```

```
|1 1 1 1.1 1|    .B T  |/./ / / /|/ / / /./|0 0 0.1 0 0 0.1 1 0|R|     MTFSB0[.]
|1 1 1 1.1 1|    .B T  |/./ / / /|/ / / /./|0 0 0.0 1 0 0.1 1 0|R|     MTFSB1[.]
|1 1 1 1.1 1|/| .    F L.M    |/|  F R B. |1 0 1.1 0 0 0.1 1 1|R|     MTFSF[.]
|1 1 1 1.1 1|  B.F|/ /|/./ / / /|    U  |/|0 0 1.0 0 0 0.1 1 0|R|     MTFSFI[.]
|0 1 1 1.1 1|    .R S  |/./ / / /|/ / / /./|0 0 1.0 0 1 0.0 1 0|/| P   MTMSR
|0 1 1 1.1 1|    .R S  | .      S.P R    . |0 1 1.1 0 1 0.0 1 1|/|     MTSPR
|0 1 1 1.1 1|    .R S  |/|  S R  |/ / / /./|0 0 1.0 1 0 1 0.0 1 0|/| P   MTSR
|0 1 1 1.1 1|    .R S  |/./ / / /|    R B. |0 0 1.1 1 1 0.0 1 0|/| P   MTSRIN        MTSRI
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|0 0.1 1 0 1.0 1 1|R|  -              MUL[O][.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |/|0 0.1 0 0 1.0 0 1|R|  ! MULHD[.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |/|0 0.0 0 0 1.0 0 1|R|  ! MULHDU[.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |/|0 0.1 0 0 1.0 1 1|R|  + MULHW[.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |/|0 0.0 0 0 1.0 1 1|R|  + MULHWU[.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|0 1.1 1 0 1.0 0 1|R|  ! MULLD[O][.]
|0 0 0 1.1 1|    .R T  | .  R A  |       .    S I.1 6    .       |     MULLI         MULI
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|0 1.1 1 0 1.0 1 1|R|     MULLW[O][.]  MULS[O][.]
|0 1 1 1.1 1|    .R T  | .  R A  |    R B. |V|1 1.1 1 0 1.0 0 0|R|  -              NABS[O][.]
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 1 1.1 0 1 1.1 0 0|R|     NAND[.]
|0 1 1 1.1 1|    .R T  | .  R A  |/ / / /./|V|0 0.1 1 0 1.0 0 0|R|     NEG[O][.]
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 0.1 1 1 1.1 0 0|R|     NOR[.]
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 1 1.0 1 1 1.1 0 0|R|     OR[.]
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 1 1.0 0 1 1.1 0 0|R|     ORC[.]
|0 1 1 0.0 0|    .R S  | .  R A  |       .    U I.1 6    .       |     ORI           ORIL
|0 1 1 0.0 1|    .R S  | .  R A  |       .    U I.1 6    .       |     ORIS          ORIU
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 1 0.0 1 1 0.0 1 0|/| P-              RAC
|0 1 0 0.1 1|/ /./ / /|/./ / / /|/ / / /./|0 0 0.0 1 1 0.0 1 0|/| P   RFI
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|0 0 0.1 0 1 0.0 1 0|/| P-              RFSVC
|0 1 1 1.1 0|    .R S  | .  R A  |    R B. |    m.b  |0 1.0 0 0|R|  ! RLDCL[.]
|0 1 1 1.1 0|    .R S  | .  R A  |    R B. |    m.e  |0 1.0 0 1|R|  ! RLDCR[.]
|0 1 1 1.1 0|    .R S  | .  R A  |    s h. |    m.b  |0 0.1 0|s|R|  ! RLDIC[.]
|0 1 1 1.1 0|    .R S  | .  R A  |    s h. |    m.b  |0 0.0 0|s|R|  ! RLDICL[.]
|0 1 1 1.1 0|    .R S  | .  R A  |    s h. |    m.e  |0 0.0 1|s|R|  ! RLDICR[.]
|0 1 1 1.1 0|    .R S  | .  R A  |    s h. |    m.b  |0 0.1 1|s|R|  ! RLDIMI[.]
|0 1 0 1.1 0|    .R S  | .  R A  |    R B. |    m.b  |   .m e  |R|  -              RLMI[.]
|0 1 0 1.0 0|    .R S  | .  R A  |    S H. |    m.b  |   .m e  |R|     RLWIMI[.]    RLIMI[.]
|0 1 0 1.0 1|    .R S  | .  R A  |    S H. |    m.b  |   .m e  |R|     RLWINM[.]    RLINM[.]
|0 1 0 1.1 1|    .R S  | .  R A  |    R B. |    m.b  |   .m e  |R|     RLWNM[.]     RLNM[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 0.0 0 1 1.0 0 1|/| -               RRIB[.]
|0 1 0 0.0 1|/ /./ / / /./ / / /./ / / /./ / / /./ / / /./ /|1|/|     SC            SVC[A]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|0 1 1.1 1 1 0.0 1 0|/| P!  SLBIA
|0 1 1 1.1 1|/ /./ / /|/./ / / /|    R B. |0 1 1.0 1 1 0.0 1 0|/| P!  SLBIE
|0 1 1 1.1 1|/ /./ / /|/./ / / /|    R B. |0 1 1.1 0 1 0.0 1 0|/| P!  SLBIEX
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 0.0 0 1 1.0 1 1|R|  ! SLD[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|0 0 1.0 0 1 1.0 0 1|/| -               SLE
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|0 0 1.1 0 1 1.0 0 1|/| -               SLEQ[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|0 0 1.0 1 1 1.0 0 0|/| -               SLIQ[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|0 0 1.1 1 1 1.0 0 0|/| -               SLLIQ[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|0 0 1.1 0 1 1.0 0 0|/| -               SLLQ[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|0 0 1.0 0 1 1.0 0 0|/| -               SLQ[.]
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 0.0 0 1 1.0 0 0|R|     SLW[.]       SL[.]
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |1 1 0.0 0 1 1.0 1 0|R|  ! SRAD[.]
|0 1 1 1.1 1|    .R S  | .  R A  |    s h. |1 0 0.1 1 1 0.1 0|s|R|  ! SRADI[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 1 1.0 1 1 1.0 0 0|/| -               SRAIQ[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 1 1.0 0 1 1.0 0 0|/| -               SRAQ[.]
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |1 1 0.0 0 1 1.0 0 0|R|     SRAW[.]      SRA[.]
|0 1 1 1.1 1|    .R S  | .  R A  |    S H. |1 1 0.0 1 1 1.0 0 0|R|     SRAWI[.]     SRAI[.]
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |1 0 0.0 0 1 1.0 1 1|R|  ! SRD[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 1.0 0 1 1.0 0 1|/| -               SRE[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 1 1.0 0 1 1.0 0 1|/| -               SREA[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 1.1 0 1 1.0 0 1|/| -               SREQ[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 1.0 1 1 1.0 0 0|/| -               SRIQ[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 1.1 1 1 1.0 0 0|/| -               SRLIQ[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 1.1 0 1 1.0 0 0|/| -               SRLQ[.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 1.0 0 1 1.0 0 0|/| -               SRQ[.]
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |1 0 0.0 0 1 1.0 0 0|R|     SRW[.]       SR[.]
|1 0 0 1.1 0|    .R S  | .  R A  |       .    D.1 6    .       |     STB
|1 0 0 1.1 1|    .R S  | .  R A  |       .    D.1 6    .       |     STBU
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 1.1 1 1 0.1 1 1|/|     STBUX
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 1.1 0 1 0.1 1 1|/|     STBX
|1 1 1 1.1 0|    .R S  | .  R A  |       .  D S 1.4      .   |0 0|  ! STD
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 1.1 0 1 0.1 1 0|1|  ! STDCX.
|1 1 1 1.1 0|    .R S  | .  R A  |       .  D S 1.4      .   |0 1|  ! STDU
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 1.0 1 1 0.1 0 1|/|  ! STDUX
|0 1 1 1.1 1|    .R S  | .  R A  |    R B. |0 0 1.0 0 1 0.1 0 1|/|  ! STDX
|1 1 0 1.1 0|  F.R S  | .  R A  |       .    D.1 6    .       |     STFD
|1 1 0 1.1 1|  F.R S  | .  R A  |       .    D.1 6    .       |     STFDU
```

```
|0 1 1 1.1 1|  F.R S  | .  R A  |    R B. |1 0 1.1 1 1 0.1 1 1|/|    STFDUX
|0 1 1 1.1 1|  F.R S  | .  R A  |    R B. |1 0 1.1 0 1 0.1 1 1|/|    STFDX
|0 1 1 1.1 1|  F.R S  | .  R A  |    R B. |1 1 1.1 0 1 0.1 1 1|/|  ? STFIWX
|1 1 0 1.0 0|  F.R S  | .  R A  |       .      D.1 6    .       |    STFS
|1 1 0 1.0 1|  F.R S  | .  R A  |       .      D.1 6    .       |    STFSU
|0 1 1 1.1 1|  F.R S  | .  R A  |    R B. |1 0 1.0 1 1 0.1 1 1|/|    STFSUX
|0 1 1 1.1 1|  F.R S  | .  R A  |    R B. |1 0 1.0 0 1 0.1 1 1|/|    STFSX
|1 0 1 1.0 0|   .R S  | .  R A  |       .      D.1 6    .       |    STH
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |1 1 1.0 0 1 0.1 1 0|/|    STHBRX
|1 0 1 1.0 1|   .R S  | .  R A  |       .      D.1 6    .       |    STHU
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |0 1 1.0 1 1 0.1 1 1|/|    STHUX
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |0 1 1.0 0 1 0.1 1 1|/|    STHX
|1 0 1 1.1 1|   .R S  | .  R A  |       .      D.1 6    .       |    STMW          STM
|0 1 1 1.1 1|   .R S  | .  R A  |    N B. |1 0 1.1 0 1 0.1 0 1|/|    STSWI         STSI
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |1 0 1.0 0 1 0.1 0 1|R|    STSWX         STSX
|1 0 0 1.0 0|   .R S  | .  R A  |       .      D.1 6    .       |    STW           ST
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |1 0 1.0 0 1 0.1 1 0|/|    STWBRX       STBRX
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |0 0 1.0 0 1 0.1 1 0|1|  + STWCX.
|1 0 0 1.0 1|   .R S  | .  R A  |       .      D.1 6    .       |    STWU          STU
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |0 0 1.0 1 1 0.1 1 1|/|    STWUX        STUX
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |0 0 1.0 0 1 0.1 1 1|/|    STWX          STX
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|0 0.0 1 0 1.0 0 0|R|  + SUBF[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|0 0.0 0 0 1.0 0 0|R|    SUBFC[O][.]   SF[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |    R B. |V|0 1.0 0 0 1.0 0 0|R|    SUBFE[O][.]   SFE[O][.]
|0 0 1 0.0 0|   .R T  | .  R A  |       .      S I.1 6    .       |    SUBFIC        SFI
|0 1 1 1.1 1|   .R T  | .  R A  |/ / / /./|V|0 1.1 1 0 1.0 0 0|R|    SUBFME[O][.] SFME[O][.]
|0 1 1 1.1 1|   .R T  | .  R A  |/ / / /./|V|0 1.1 0 0 1.0 0 0|R|    SUBFZE[O][.] SFZE[O][.]
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|1 0 0.1 0 1 0.1 1 0|/|    SYNC        DCS
|0 1 1 1.1 1|   .T O  | .  R A  |    R B. |0 0 0.1 0 0 0.1 0 0|/|  ! TD
|0 0 0 0.1 0|   .T O  | .  R A  |       .      S I.1 6    .       |  ! TDI
|0 1 1 1.1 1|/ /./ / /|/./ / / /|/ / / /./|0 1 0.1 1 1 0.0 1 0|/| P? TLBIA
|0 1 1 1.1 1|/ /./ / /|/./ / / /|    R B. |0 1 0.0 1 1 0.0 1 0|/| P? TLBIE        TLBI
|0 1 1 1.1 1|/ /./ / /|/./ / / /|    R B. |0 1 0.1 0 1 0.0 1 0|/| P? TLBIEX
|0 1 1 1.1 1|   .T O  | .  R A  |    R B. |0 0 0.0 0 0 0.1 0 0|/|    TW          T
|0 0 0 0.1 1|   .T O  | .  R A  |       .      S I.1 6    .       |    TWI           TI
|0 1 1 1.1 1|   .R S  | .  R A  |    R B. |0 1 0.0 1 1 1.1 0 0|R|    XOR[.]
|0 1 1 0.1 0|   .R S  | .  R A  |       .      U I.1 6    .       |    XORI          XORIL
|0 1 1 0.1 1|   .R S  | .  R A  |       .      U I.1 6    .       |    XORIS         XORIU
```